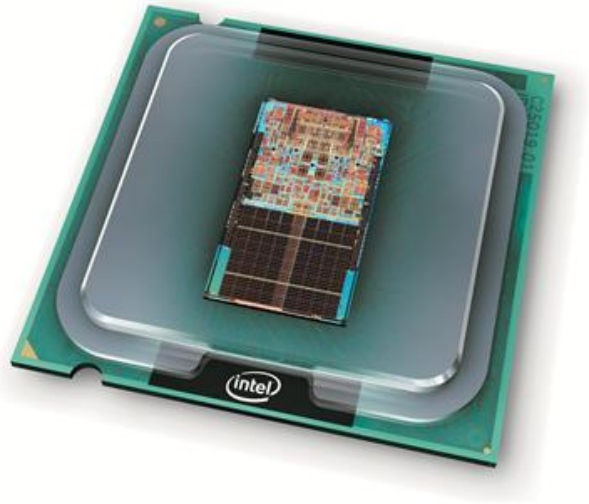


La Pila en los Procesadores IA-32 e Intel©64

Alejandro Furfaro
Ilustraciones de David Gonzalez Marquez (tnx a lot)

13 de abril de 2020

Agenda

Funcionamiento básico

Pila

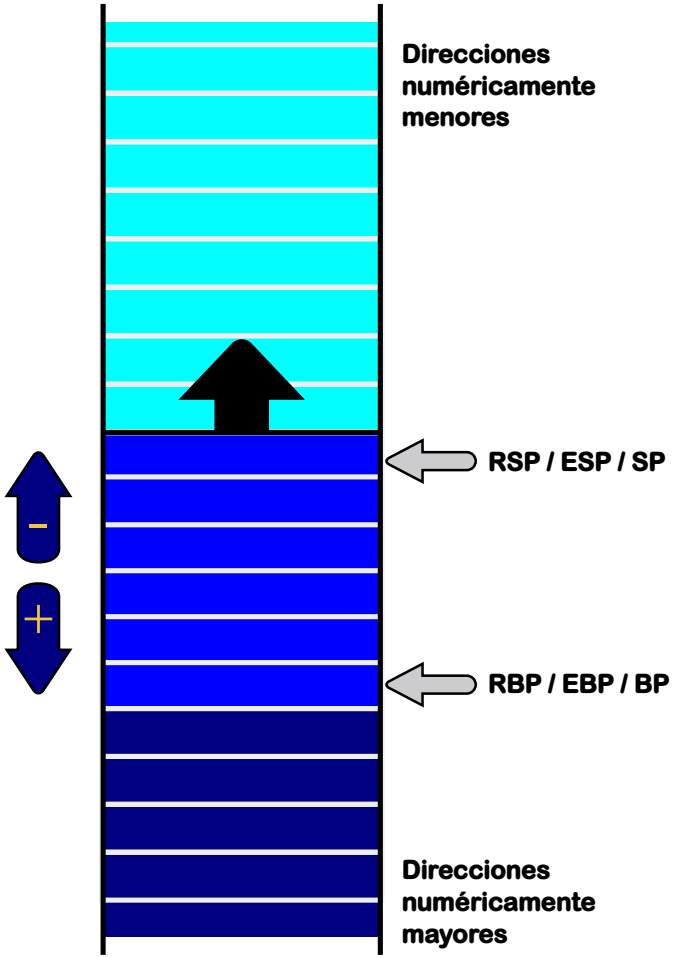


Figura: Funcionamiento Básico de la Pila.©David

- La pila (stack) es un área de memoria contigua, referenciada por un segmento cuyo selector está siempre en el registro SS del procesador.
- El tamaño de este segmento en el modo IA-32, puede llegar hasta 4 Gbytes de memoria, en especial cuando el sistema operativo utiliza el modelo de segmentación Flat (como veremos en clases subsiguientes).
- El segmento se recorre mediante un registro de propósito general, denominado habitualmente en forma genérica stack pointer, y que en estos procesadores según el modo de trabajo es el registro SP, ESP, o RSP (16, 32, o 64 bits respectivamente).

Funcionamiento básico

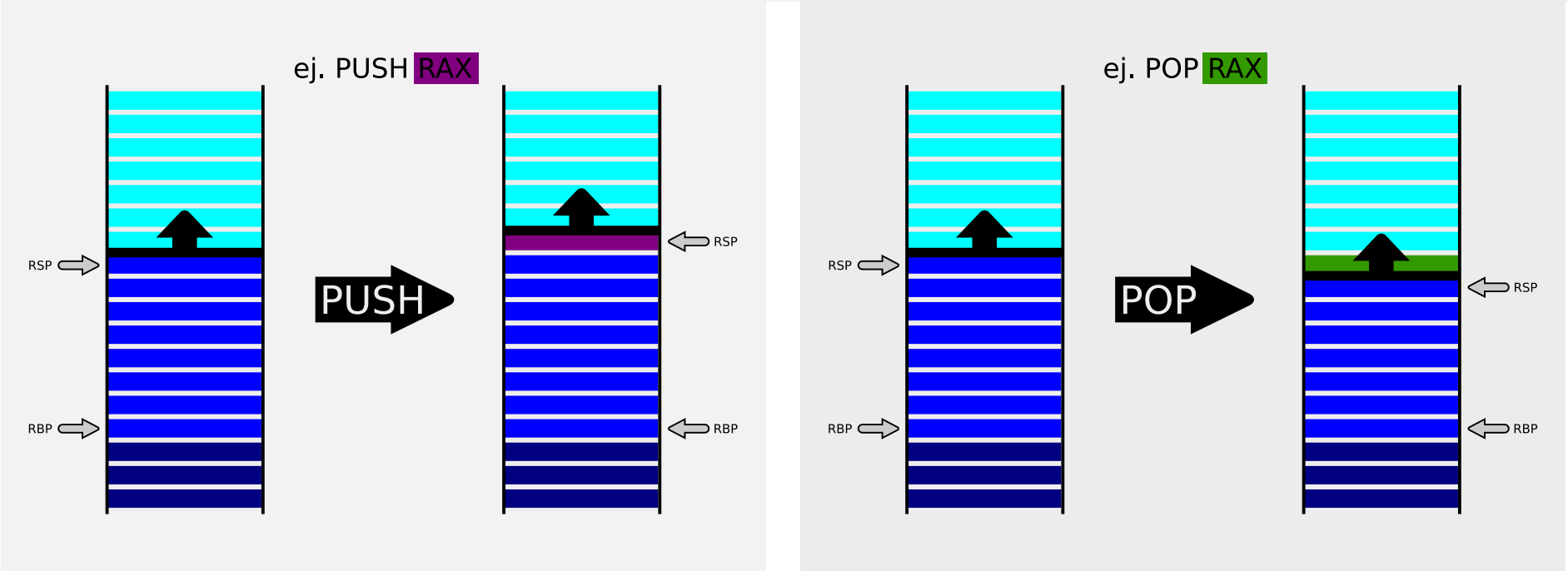


Figura: PUSH y POP.©David

- Para guardar un dato en el stack el procesador tiene la instrucción PUSH, y para retirarlo, la instrucción POP.
- Cada vez que ejecuta PUSH, el procesador decrementa el stack pointer (SP, ESP, o RSP) y luego escribe el dato en el stack, en la dirección apuntada por el registro de segmento SS, y el stack pointer correspondiente al modo de trabajo.
- Cada vez que ejecuta un POP, el procesador lee el ítem apuntado por el par SS : stack pointer, y luego incrementa éste último registro.

Primeras conclusiones

El stack es un segmento expand down, ya que a medida que lo utilizamos (PUSH) su registro de desplazamiento se decrementa apuntando a las direcciones mas bajas (down) de memoria, es decir a aquellas numéricamente menores.

Cuando se utiliza el stack

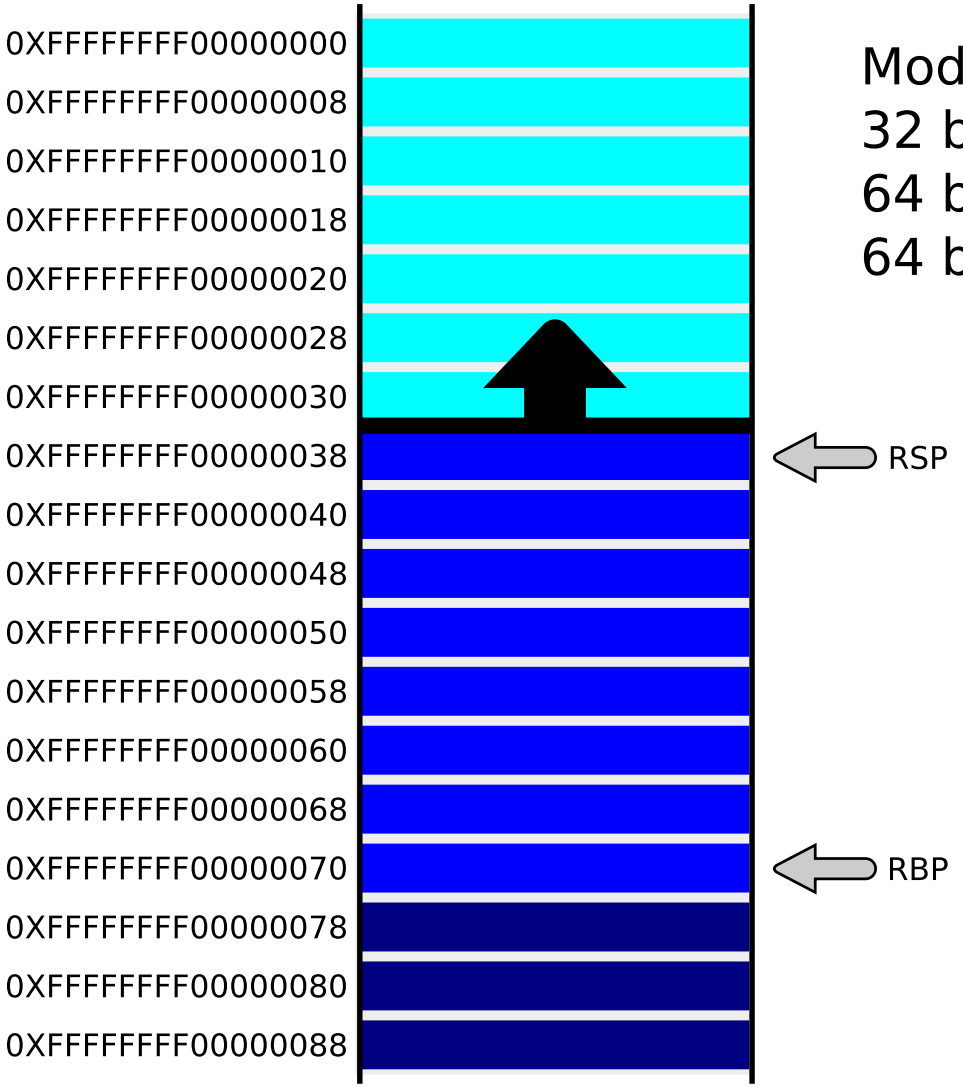
Las operaciones de pila se pueden realizar en cualquier momento, pero hablando mas generalmente, podemos afirmar que la pila se usa cuando:

- Cuando llamamos a una subrutina desde un programa en Assembler, mediante la instrucción CALL.
- Cuando el hardware mediante la interfaz adecuada envía una Interrupción al Procesador.
- Cuando desde una aplicación, ejecutamos una Interrupción de software mediante la instrucción INT type.
- Cuando desde un lenguaje como el C se invoca a una función cualquiera.

Alineación del Stack

- El stack pointer debe apuntar a direcciones de memoria alineadas de acuerdo con su ancho de bits.
- Por ejemplo, el ESP (32 bits) debe estar alineado a double words.
- Al definir un stack en memoria se debe cuidar el detalle de la alineación.
- El tamaño de cada elemento de la pila se corresponde con el atributo de tamaño del segmento (16, 32, o 64 bits), es decir, con el modo de trabajo en el que está el procesador, y no con el del operando en sí.
- Ej: PUSH AL, consume 16, 32, o 64 bits dependiendo del tamaño del segmento. Nunca consume 8 bits.
- El valor en que se decrementa el Stack Pointer se corresponde con el tamaño del segmento (2, 4, u 8 bytes).

Alineación del Stack



Modo	Alineación
32 bits	4 bytes (ABI32)
64 bits	8 bytes (ABI64)
64 bits	16 bytes (libc)

Instrucciones

- PUSH
- POP

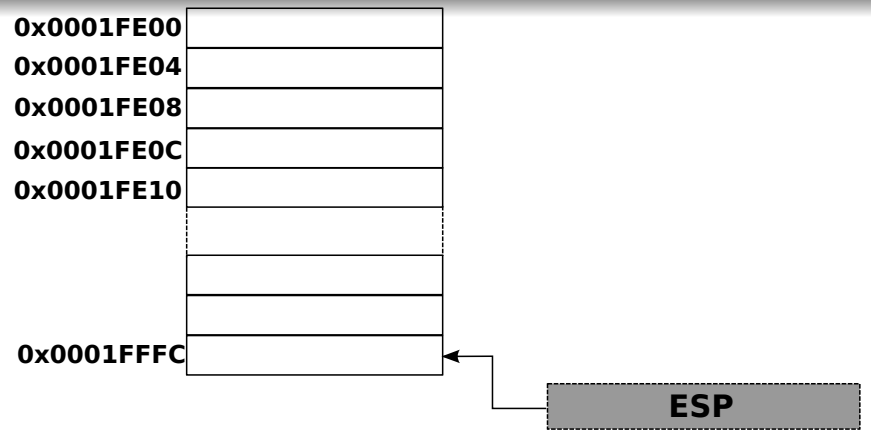
Registros

- RSP
- RBP

Figura: Alineación según el modo de trabajo.©David

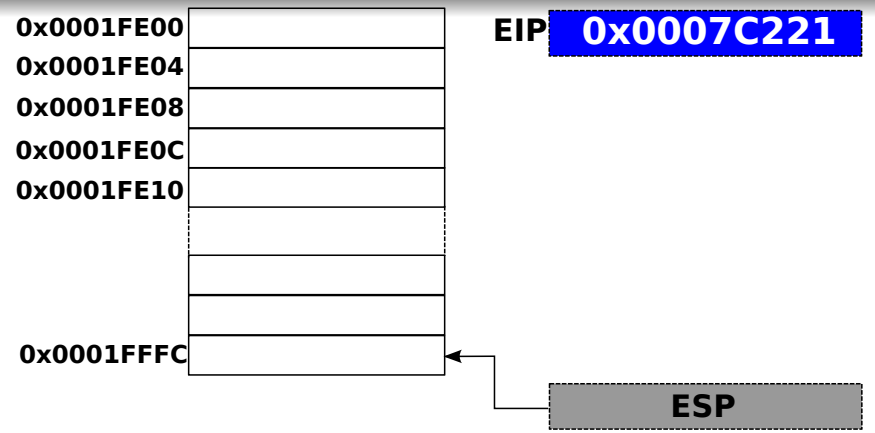
Como en un debuger :)

- Ejecutamos la primer instrucción
- Lee el port de E/S
- Y luego.....




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
in ax,dx ;lee port
   call setmask ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask    ;aplica la mascara
   ret            ;retorna
```

Estamos a punto de ejecutar CALL

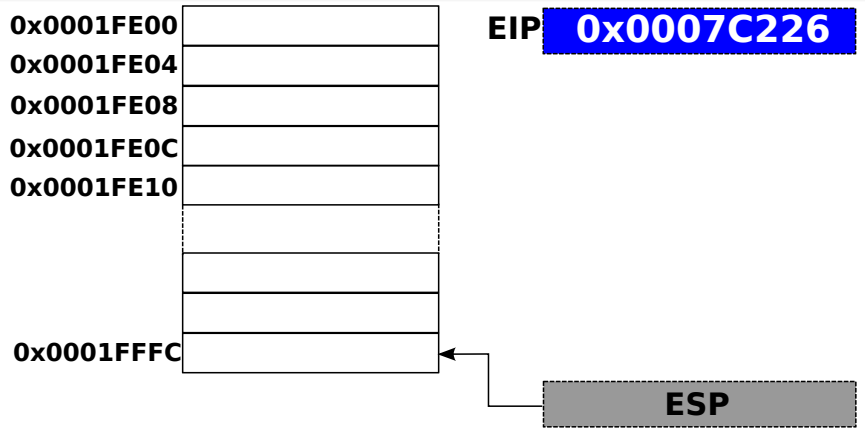


- ...ejecutamos la instrucción Call.
- La misma está almacenada a partir de la dirección de memoria contenida por **EIP**.
- El ESP apunta a la base de la pila.

```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
call setmask ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask       ;aplica la mascara
   ret               ;retorna
```

CALL por dentro...

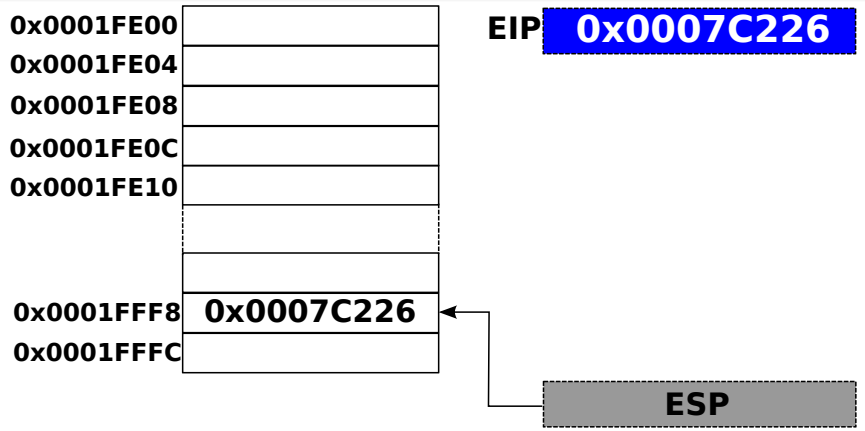
- En primer lugar el procesador apunta con EIP a la siguiente instrucción.
- Un CALL near se compone de 1 byte de código de operación y cuatro bytes para la dirección efectiva (offset), ya que estamos en 32 bits.
- Por eso el EIP apunta 5 bytes mas adelante, ya que allí comienza la siguiente instrucción del CALL.




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,x        ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask     ;aplica la mascara
   ret             ;retorna
```

CALL por dentro...

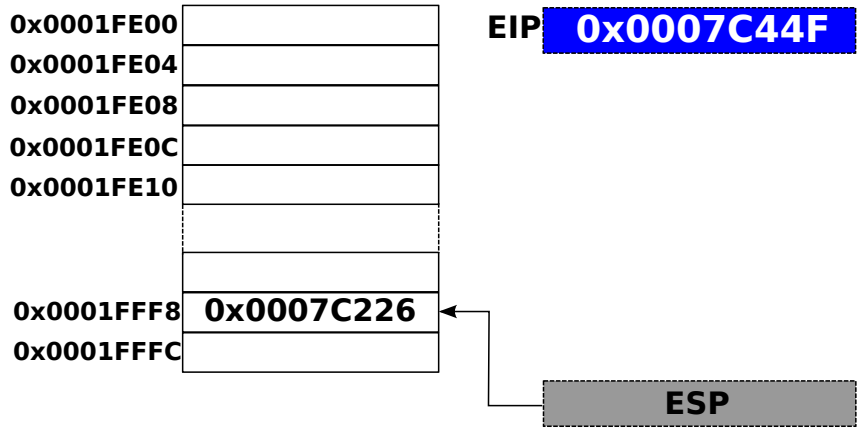
- El procesador decrementa ESP y guarda el valor de EIP.
- Así resguarda su dirección de retorno a la instrucción siguiente a CALL.
- Para saber a donde debe saltar saca de la instrucción CALL la dirección efectiva de la subrutina *setmask*.
- En nuestro caso 0x0007C44F.




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask       ;aplica la mascara
   ret               ;retorna
```

Resultado del CALL

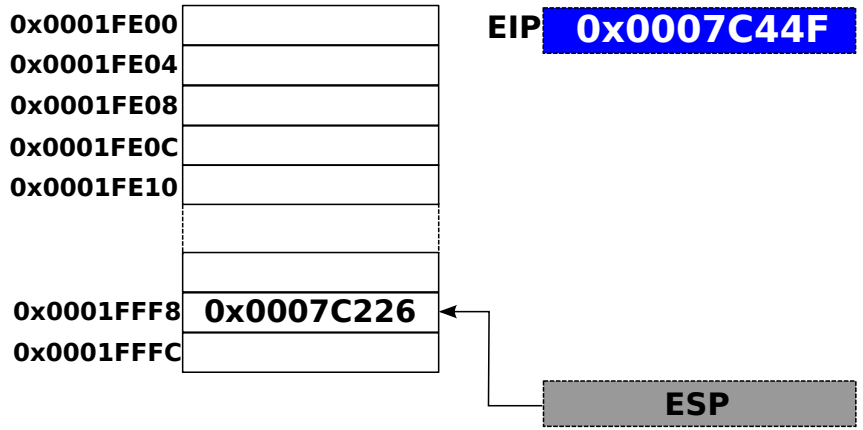
- Como resultado el valor de EIP, es reemplazado por la dirección efectiva de la subrutina *setmask*.
- Y sin mas.... el procesador está buscando la primer instrucción de la subrutina *setmask*, en este caso, la operación and.




```
%define mask    0xfff0
main:
    ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
    ...
    ...
setmask:
and ax,mask ;aplica la mascara
   ret               ;retorna
```

Volver.....

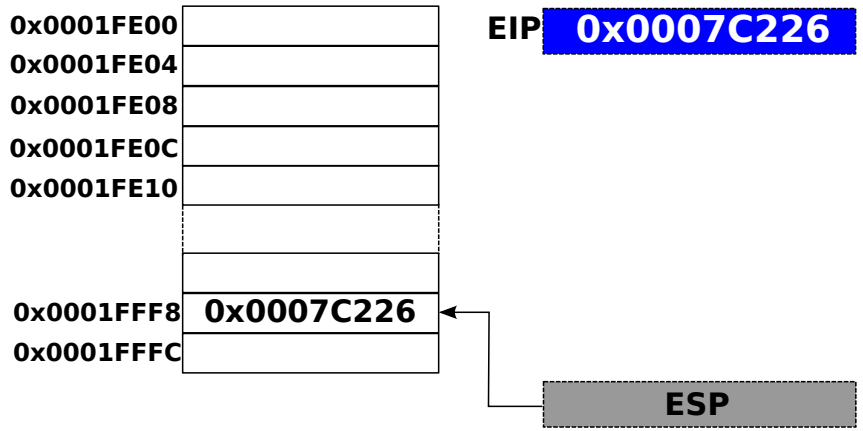
- Esta subrutina es trivial a los efectos del ejemplo.
- Para volver (sin la frente marchita)...
- Es necesario **ret**ornar




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask     ;aplica la mascara
ret ;retorna
```

Volviendo.....

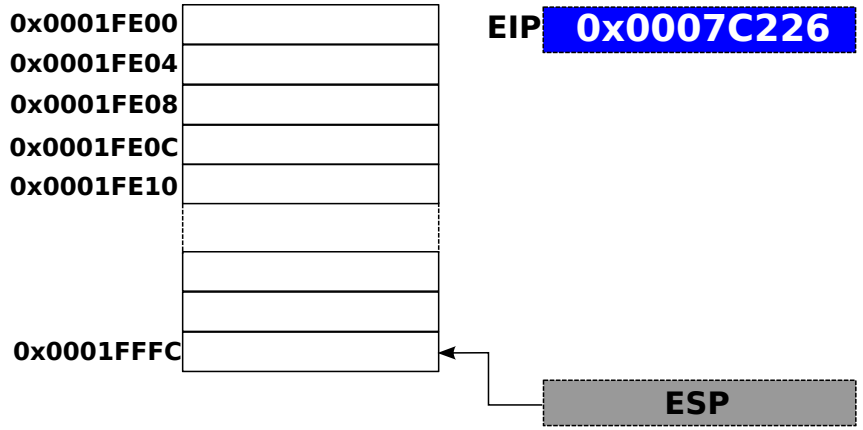
- La ejecución de **ret** consiste en recuperar de la pila la dirección de retorno.
- Esa dirección se debe cargar en EIP
- Una vez hecho.....




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
   ...
   ...
setmask:
   and ax,mask     ;aplica la mascara
   ret             ;retorna
```

Volvimos!

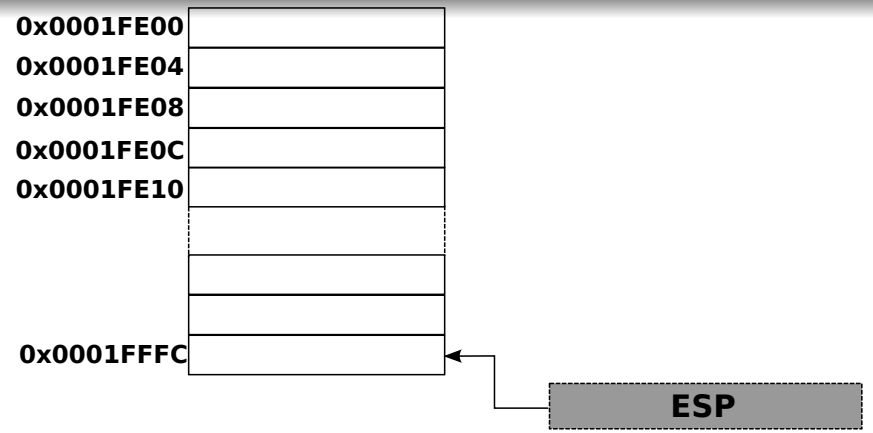
- Finalizada la ejecución de **ret** estamos otra vez en el código llamador.
- Pero en la instrucción siguiente a CALL




```
%define mask    0xfff0
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  setmask  ;llama a subrutina para aplicar una mascara
...
   ...
setmask:
   and ax,mask     ;aplica la mascara
   ret             ;retorna
```

Ahora el destino está en un segmento diferente

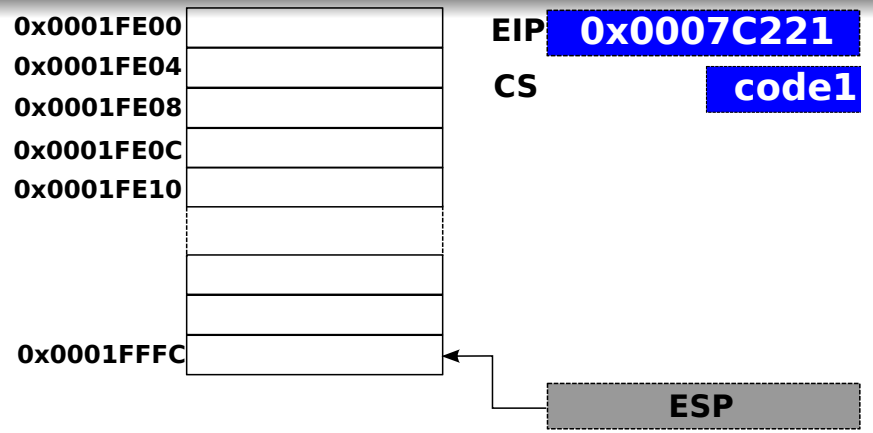
- Ejecutamos la primer instrucción
- Lee el port de E/S
- Y luego.....




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
in ax,dx ;lee port
   call   code2:setmask     ;llama a aplicar m scara
   ...
section code2
setmask:
   and    ax,mask           ;aplica la m scara
   retf                     ;retorna
```

Por lo tanto importa el valor de CS...

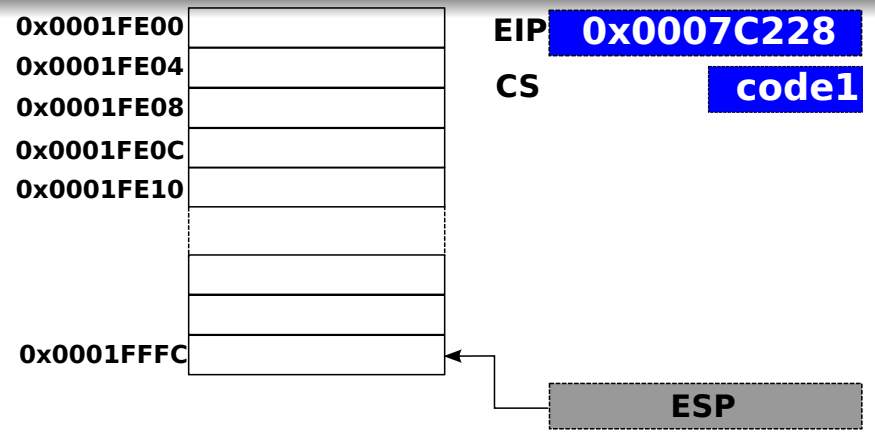
- Nuevamente nos paramos en el CALL
- Pero ahora necesitamos memorizar EIP, y también CS
- Ya que al estar el destino en otro segmento CS se modificará




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
call code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask             ;aplica la mscara
   retf                      ;retorna
```

Otra vez adentro del CALL... pero FAR

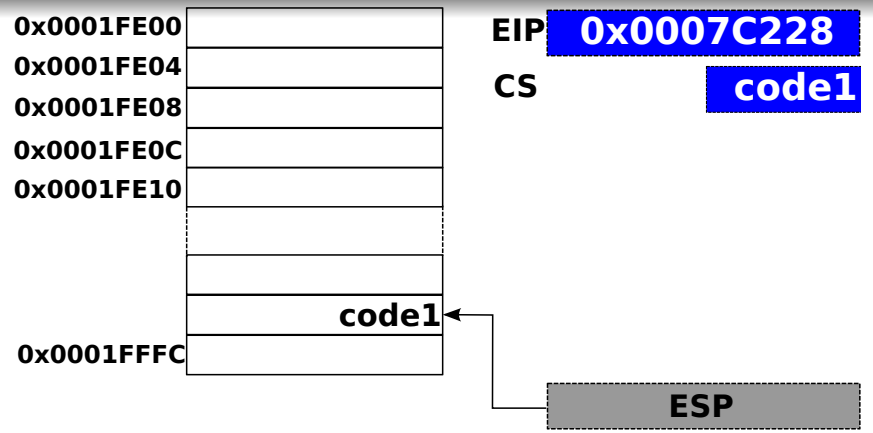
- Ahora la instrucción mide 7 bytes ya que se agrega el segmento
- Por lo tanto el EIP se incrementa 7 lugares
- Y se memoriza en la pila la dirección FAR.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask           ;aplica la mscara
   retf                    ;retorna
```

Otra vez adentro del CALL... pero FAR

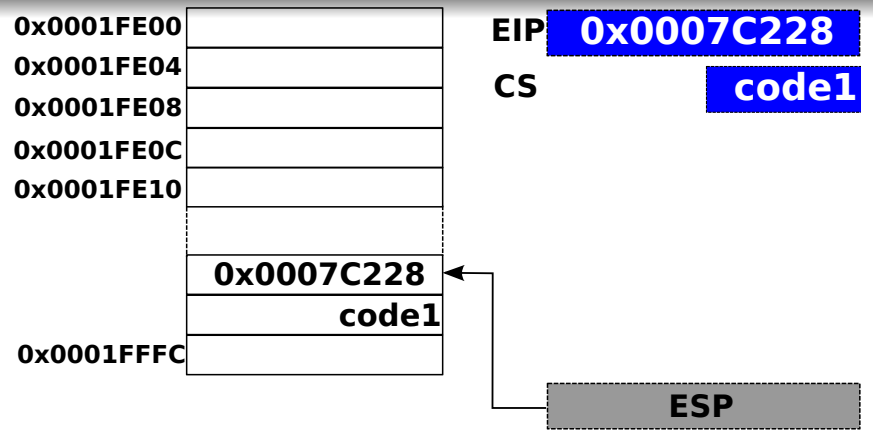
- En primer lugar guarda en la pila, el valor del segmento al cual debe retornar.
- Siempre antes de almacenar nada en la pila, debe antes decrementar el valor del ESP.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask           ;aplica la mscara
   retf                    ;retorna
```

Otra vez adentro del CALL... pero FAR

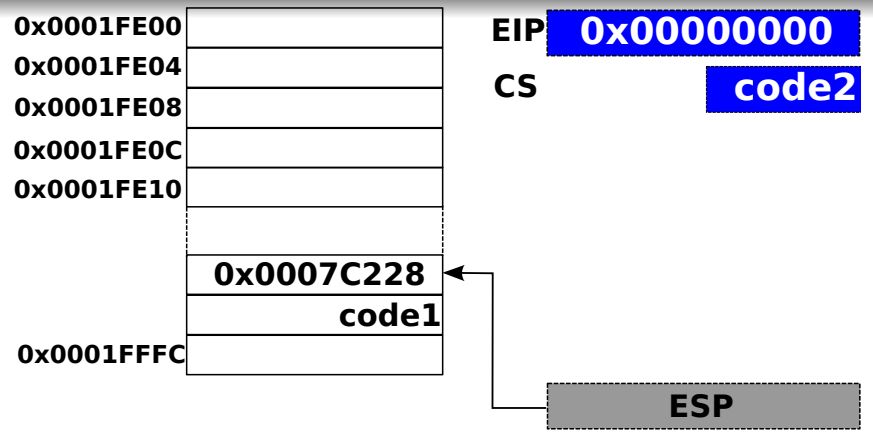
- Luego del valor del segmento guarda en la pila, el valor de EIP al cual debe retornar, y que lo llevará a buscar la siguiente instrucción al CALL.
- Decrementará nuevamente el valor del ESP, antes de almacenar.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask           ;aplica la mscara
   retf                    ;retorna
```

Otra vez adentro del CALL... pero FAR

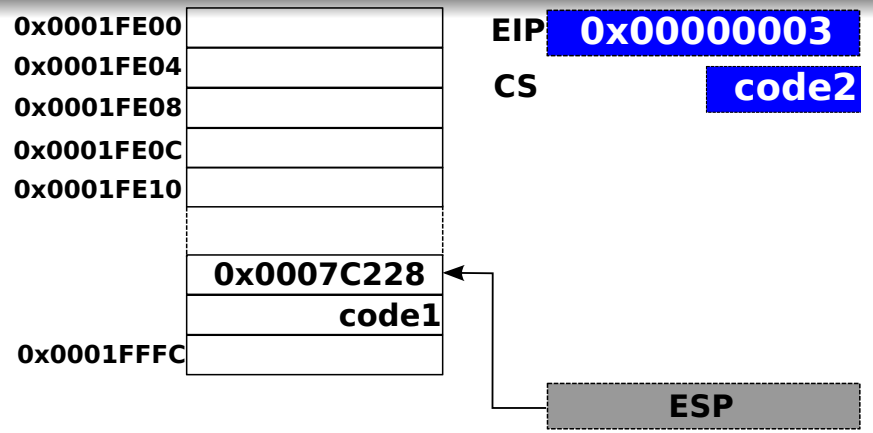
- La dirección de la rutina *setmask*, ahora es code2:offset.
- Como comienza justo al inicio del segmento, su offset es 0x00000000.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
and ax,mask ;aplica la máscara
   retf                        ;retorna
```

Retornando de un Call Far

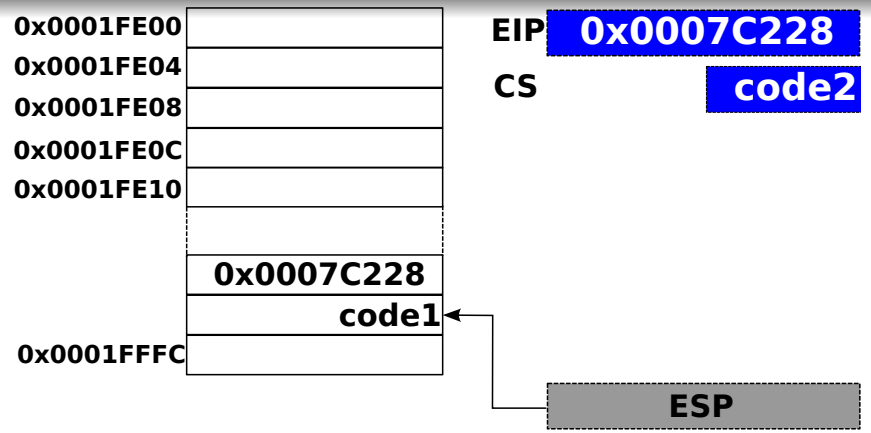
- Para volver de un call far hay que sacar de la pila no solo el offset sino también el segmento.
- Entonces no sirve la misma instrucción que se usa para volver de una rutina Near.




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask           ;aplica la mascara
retf ;retorna
```

Retornando de un Call Far

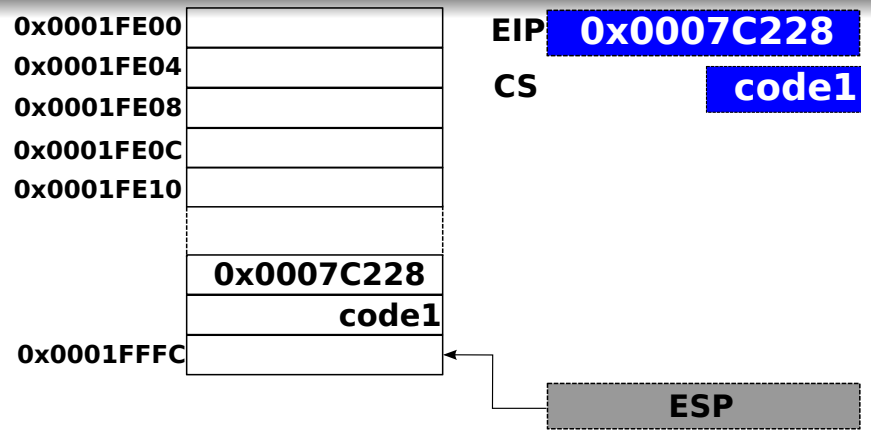
- Recupera la dirección efectiva
- Luego decrementa el Stack Pointer




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,0x300  ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
   ...
section code2
setmask:
   and   ax,mask           ;aplica la mscara
   retf                    ;retorna
```

Retornando de un Call Far

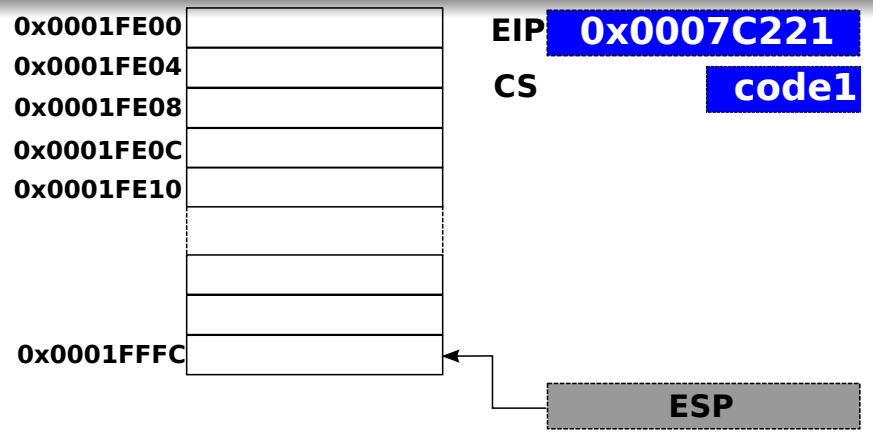
- Recupera el valor del segmento
- Luego decrementa el Stack Pointer
- ...y volvió...




```
%define mask    0xfff0
section code1
main:
   ...
   mov   dx,0x300
   in    ax,dx       ;lee port
   call  code2:setmask ;llama a subrutina para aplicar una mascara
...
section code2
setmask:
   and   ax,mask           ;aplica la mscara
   retf  ;retorna
```

¿Que pasa cuando se produce una Interrupción?

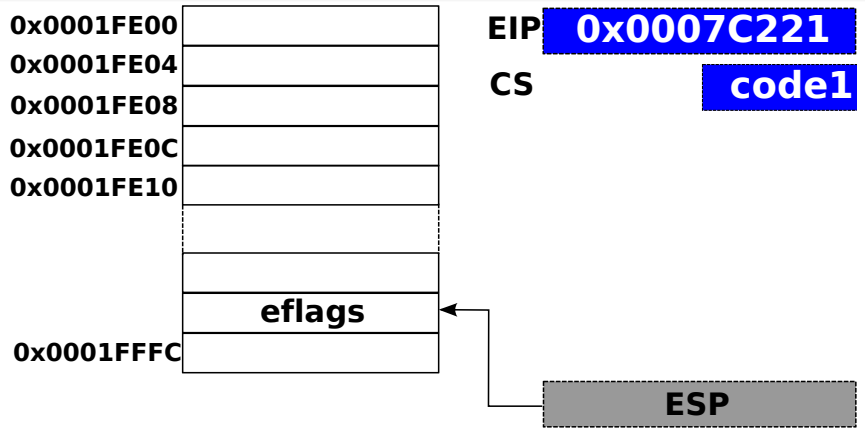
- Ejecutamos una instrucción cualquiera
- y en el medio de esa instrucción se produce una interrupción
- ...




```
section code
main:
   ...
next:
test [var],1 ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

¿Que pasa cuando se produce una Interrupción?

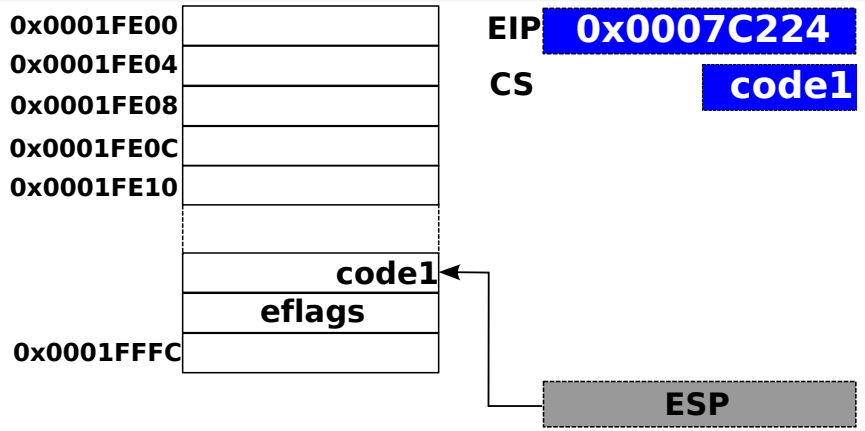
- Es necesario guardar además de la dirección de retorno, el estado del procesador.
- De otro modo si al final de la interrupción alguna instrucción modifica un flag, el estado de la máquina se altera y le vuelve al programa modificado.
- Esto puede tener resultados impredecibles si al retorno hay que usar el flag que cambió.




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

¿Que pasa cuando se produce una Interrupción?

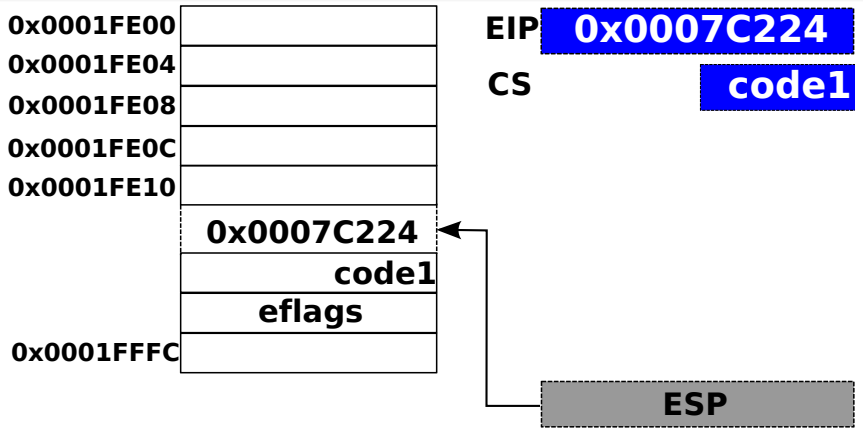
- La dirección de retorno es far.
- Especialmente en sistemas multitasking donde cada proceso tiene una pila de kernel diferente.
- Así que luego de los flags se guarda el segmento de código.




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

¿Que pasa cuando se produce una Interrupción?

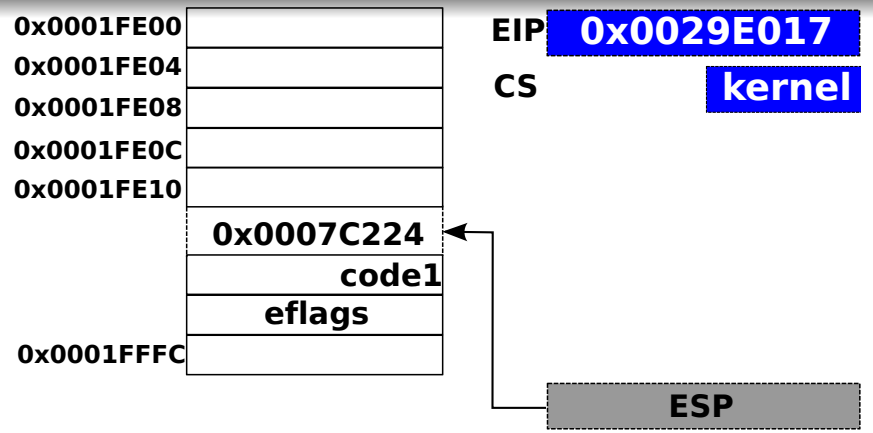
- Se resguarda finalmente la dirección efectiva
- Notar que es la de la instrucción siguiente a la de la interrupción




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

¿Que pasa cuando se produce una Interrupción?

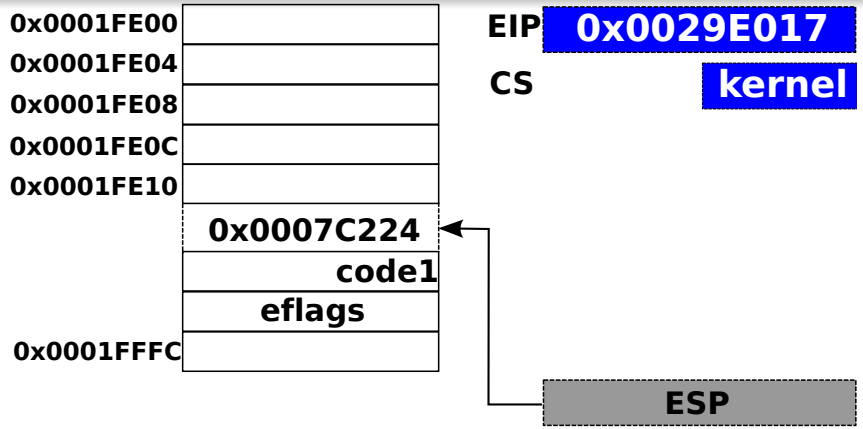
- Los nuevos valores de segmento y desplazamiento que debe cargar en CS:EIP, los obtiene del vector de interrupciones en modo real, o de la Tabla de descriptores de interrupción en modo protegido, o en el modo 64 bits.




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
in al,port ;lee port de E/S
   iret            ;retorna
```

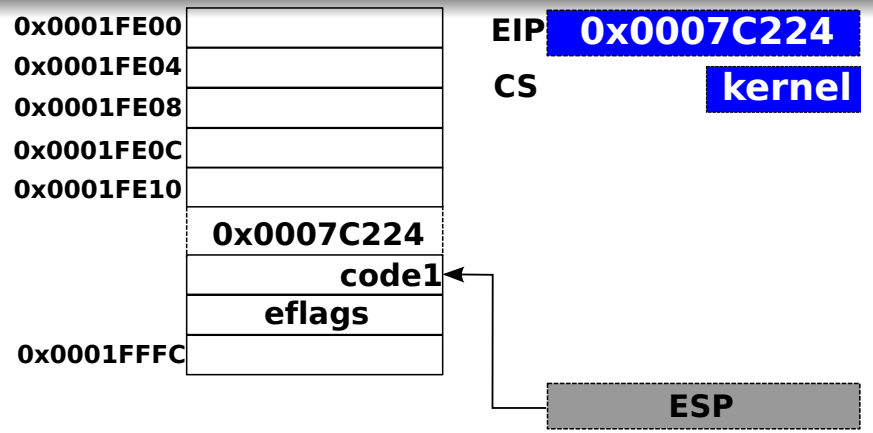
¿Como se vuelve de una Interrupción?

- Recuperando además de la dirección de retorno, los flags
- Por lo tanto necesitamos otra instrucción particular de retorno...
- ... **iret**...



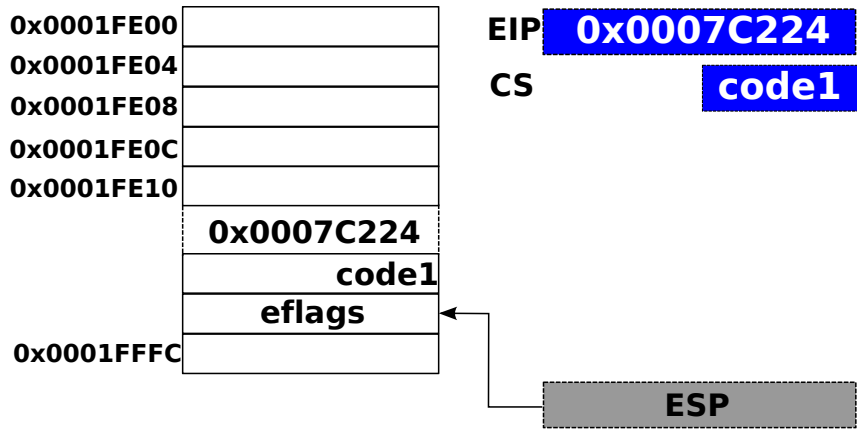

```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
iret ;retorna
```

Volviendo...



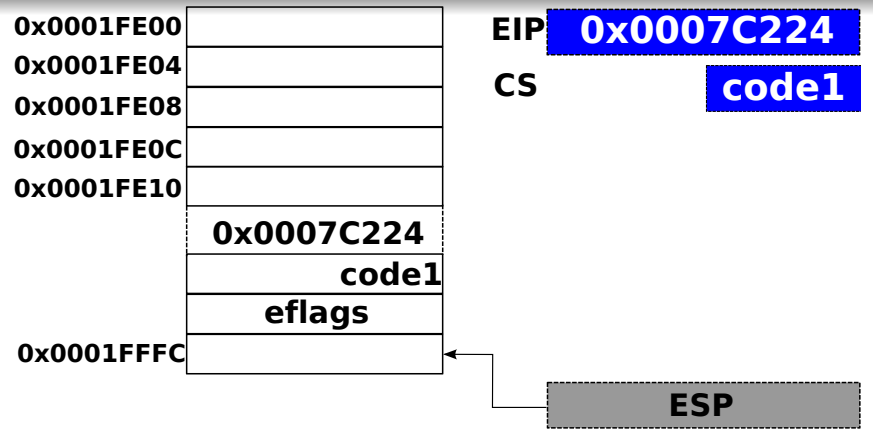

```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

Volviendo...




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
   jnz    next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

Volviendo...




```
section code
main:
   ...
next:
   test   [var],1  ;chequea bit 0 de variable
jnz next
   ...
section kernel
handler_int:
   in     al,port  ;lee port de E/S
   iret            ;retorna
```

Llamadas a función

- En general en el lenguaje C una función se invoca de la siguiente forma

```
type function (arg1, arg2,..., argn);
```

- ***type***, es siempre un tipo de dato básico (int, char, float, double), o un puntero, o void en caso en que no devuelva nada.
- El manejo de la interfaz entre el programa invocante y la función llamada la resuelve el compilador, de una manera perfectamente definida.
- Sin embargo los pormenores son diferentes según se trabaje en 32 bits o en 64 bits

Stack Frame

- El compilador traduce el llamado en el siguiente código assembler:

```
push   argn
...
push  arg2
push  arg1
call  function          ; o sea un CALL Near!!
```

- Los argumentos se apilan desde la derecha hacia la izquierda.
- Una vez dentro de la subrutina “function”, el compilador agrega el siguiente código

```
push   ebp              ;resguardamos ebp
mov    ebp,esp          ;lo apuntamos a la pila
```

Ejemplo en 32 bits

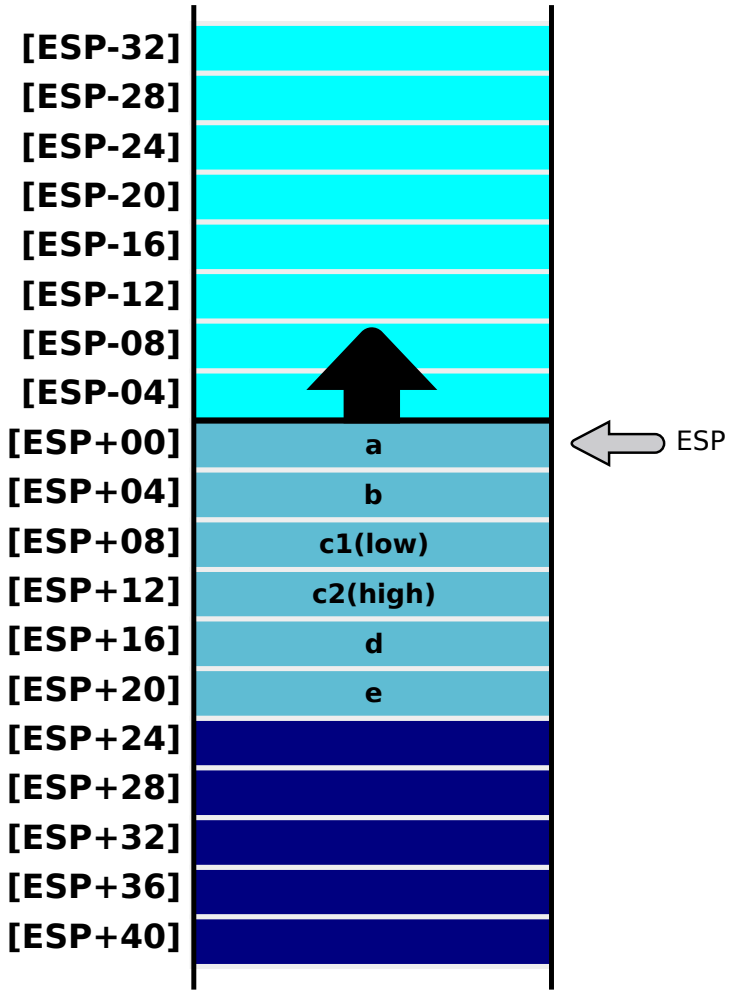
```
int f1( int a, float b, double c, int* d, double* e)
```

Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

Llamando:

```
.....
push    e
push    d
push    c2
push    c1
push    b
push    a
call    funcion
add     esp,6*4
```

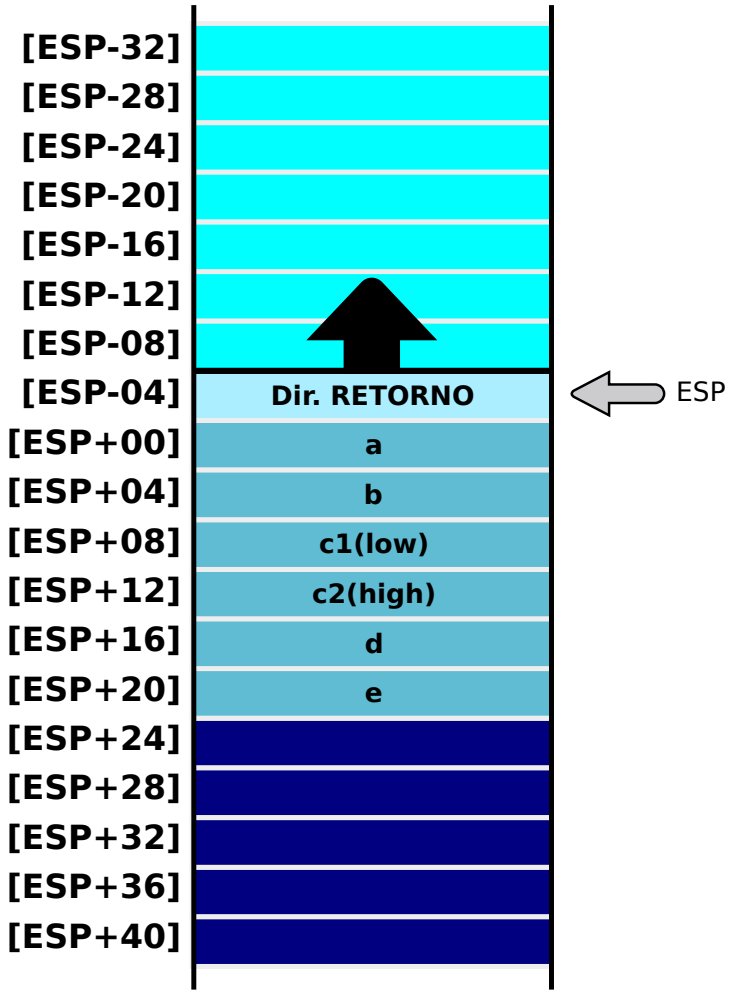


Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

Llamando:

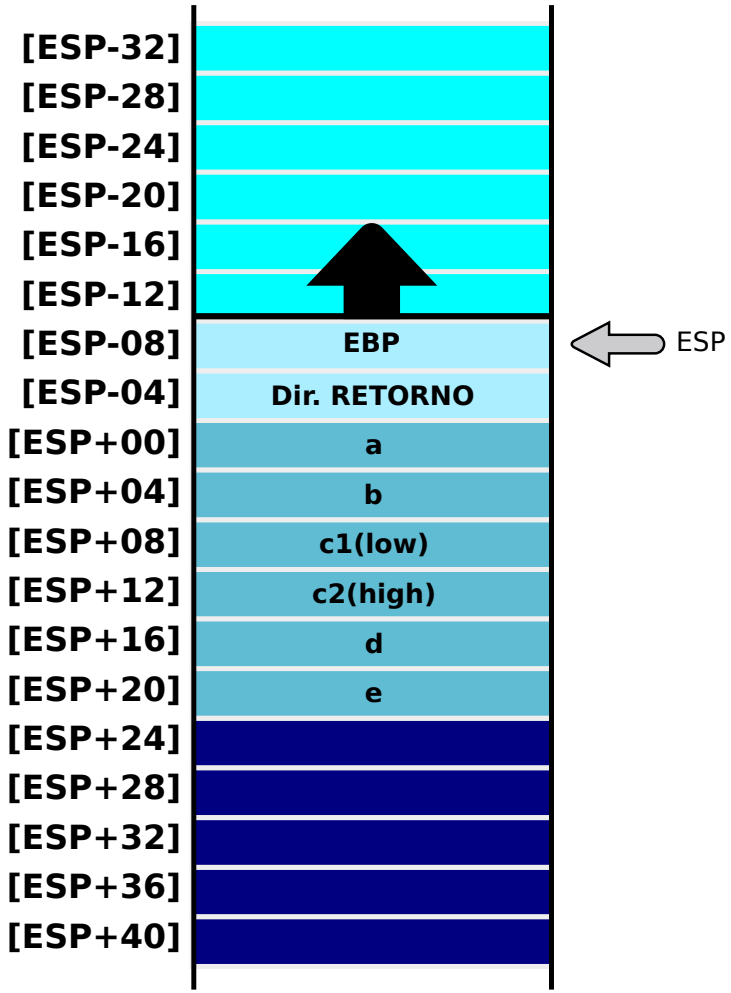
```
.....
push    e
push    d
push    c2
push    c1
push    b
push    a
call    funcion
add     esp,6*4
```



Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

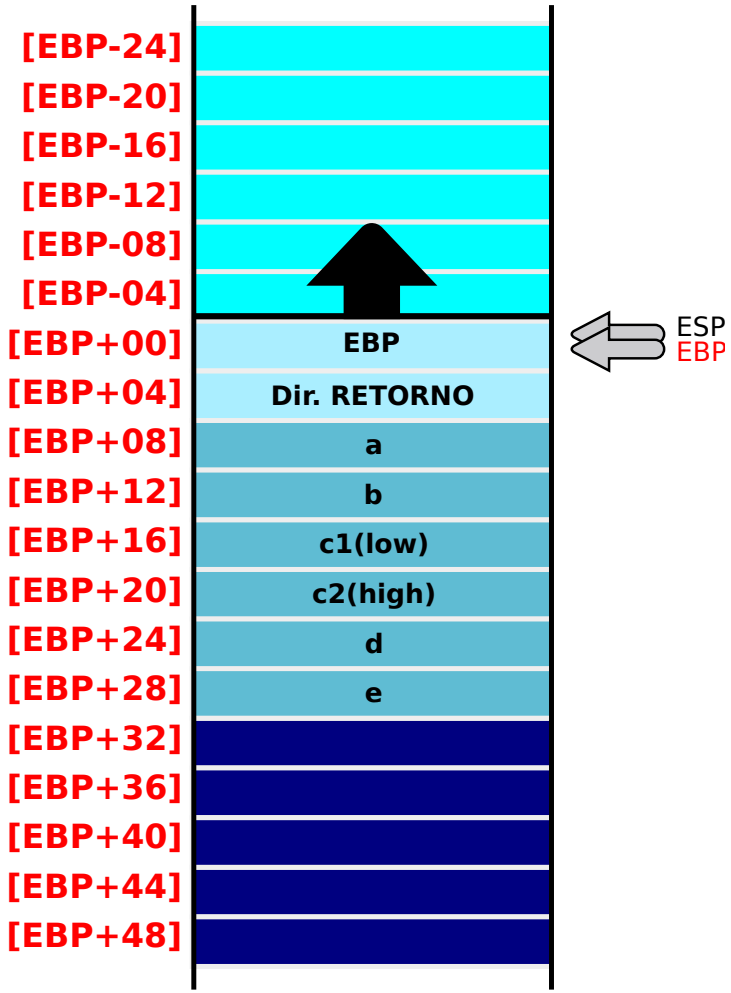
```
funcion:
    push    ebp
    mov     ebp,esp
    sub     esp,3*4
```



Ejemplo en 32 bits

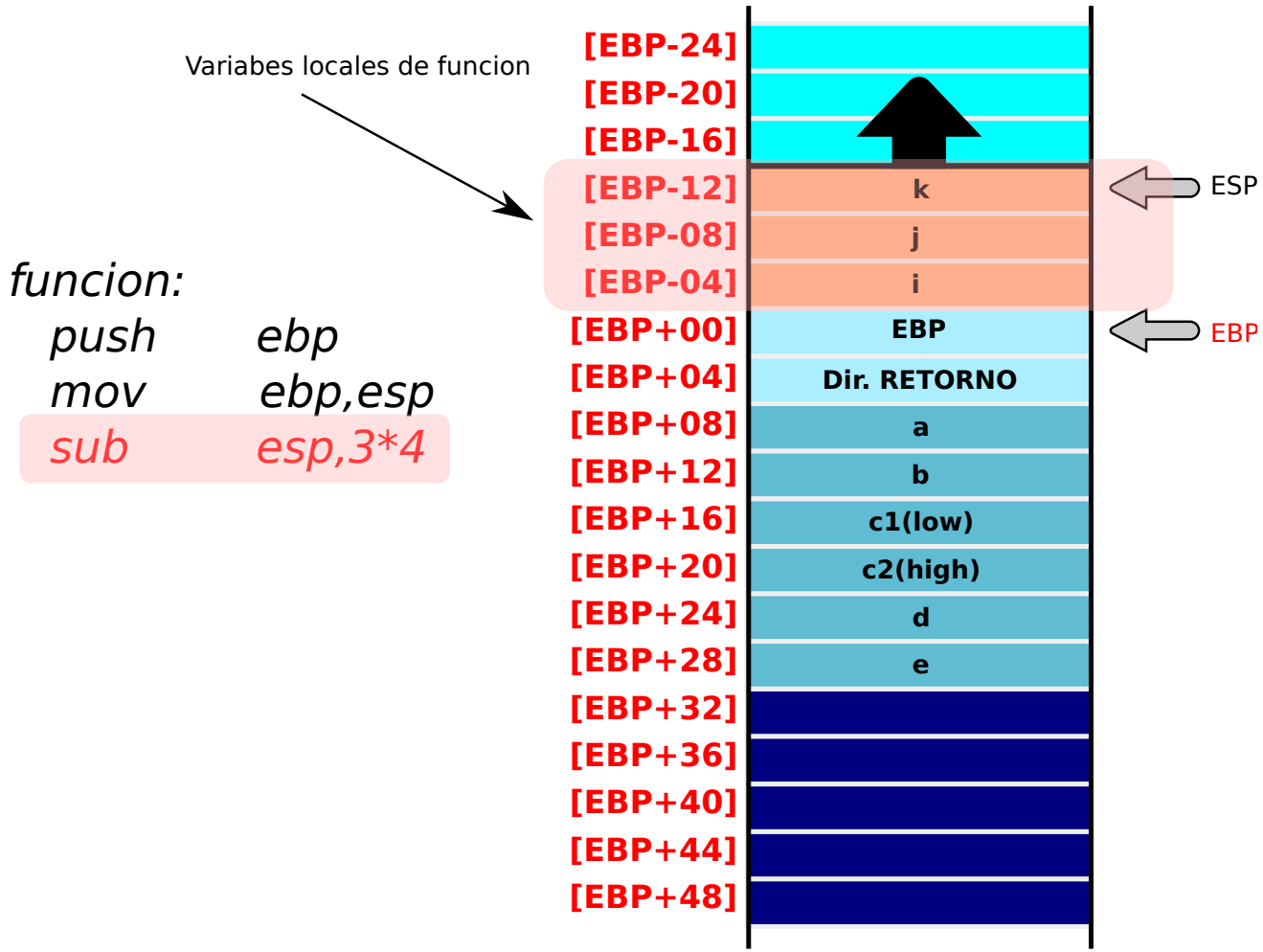
```
int f1( int a, float b, double c, int* d, double* e)
```

```
funcion:
  push    ebp
  mov     ebp,esp
  sub     esp,3*4
```



Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

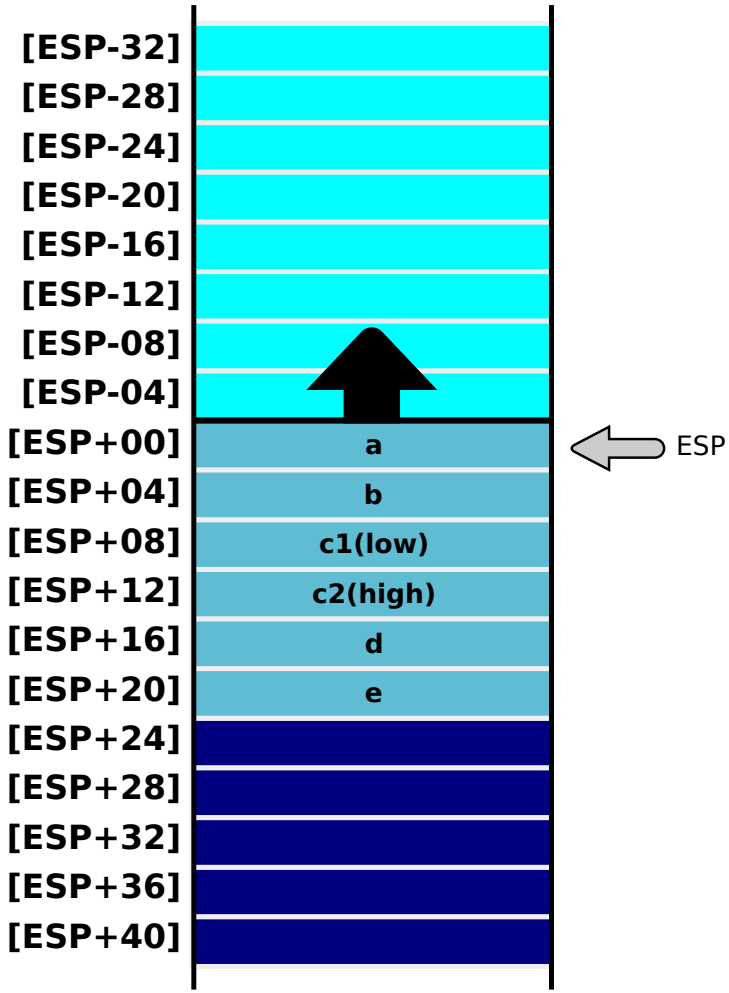


Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

Llamando:

```
.....
push    e
push    d
push    c2
push    c1
push    b
push    a
call    funcion
add     esp,6*4
```

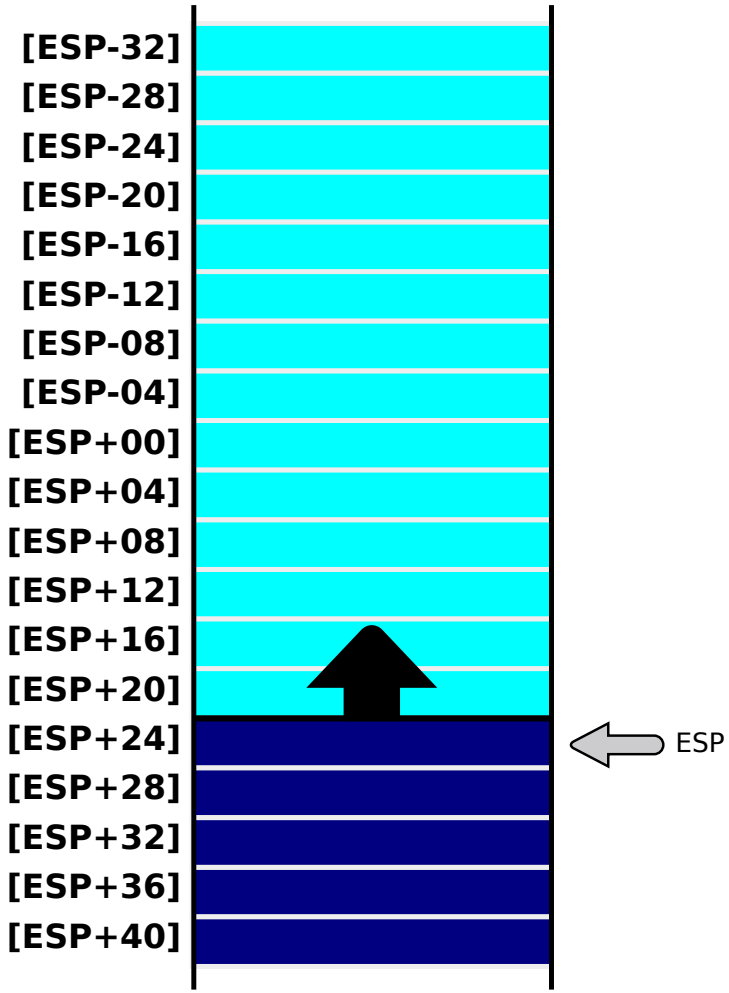


Ejemplo en 32 bits

```
int f1( int a, float b, double c, int* d, double* e)
```

Llamando:

```
.....
push    e
push    d
push    c2
push    c1
push    b
push    a
call    funcion
add     esp,6*4
```



Llamar a funciones ASM desde C

Hacemos uso de la cláusula extern en C y global en ASM:

funcion.asm

```
global fun
section .text
fun:
...
...
ret
```

programa.c

```
extern int fun(int, int);
int main(){
    ...
    fun(44,3);
    ..
}
```

Primero ensamblamos y compilamos el código en ASM para luego linkearlo con el código en C:

- `nasm -f elf64 funcion.asm -o funcion.o`
- `gcc -o ejec programa.c funcion.o`

Llamar a funciones ASM desde C

Hacemos uso de la cláusula extern en C y global en ASM:

funcion.asm

```
global fun
section .text
fun:
...
...
ret
```

programa.c

```
extern int fun(int, int);
int main(){
    ...
    fun(44,3);
    ..
}
```

Primero ensamblamos y compilamos el código en ASM para luego linkearlo con el código en C:

- `nasm -f elf64 funcion.asm -o funcion.o`
- `gcc -o ejec programa.c funcion.o`

Llamar funciones C desde ASM

Usamos sólo la cláusula extern en ASM:

main.asm

```
global main
extern fun
section .text
main:
    ...
    call fun
    ...
    ret
```

funcion.c

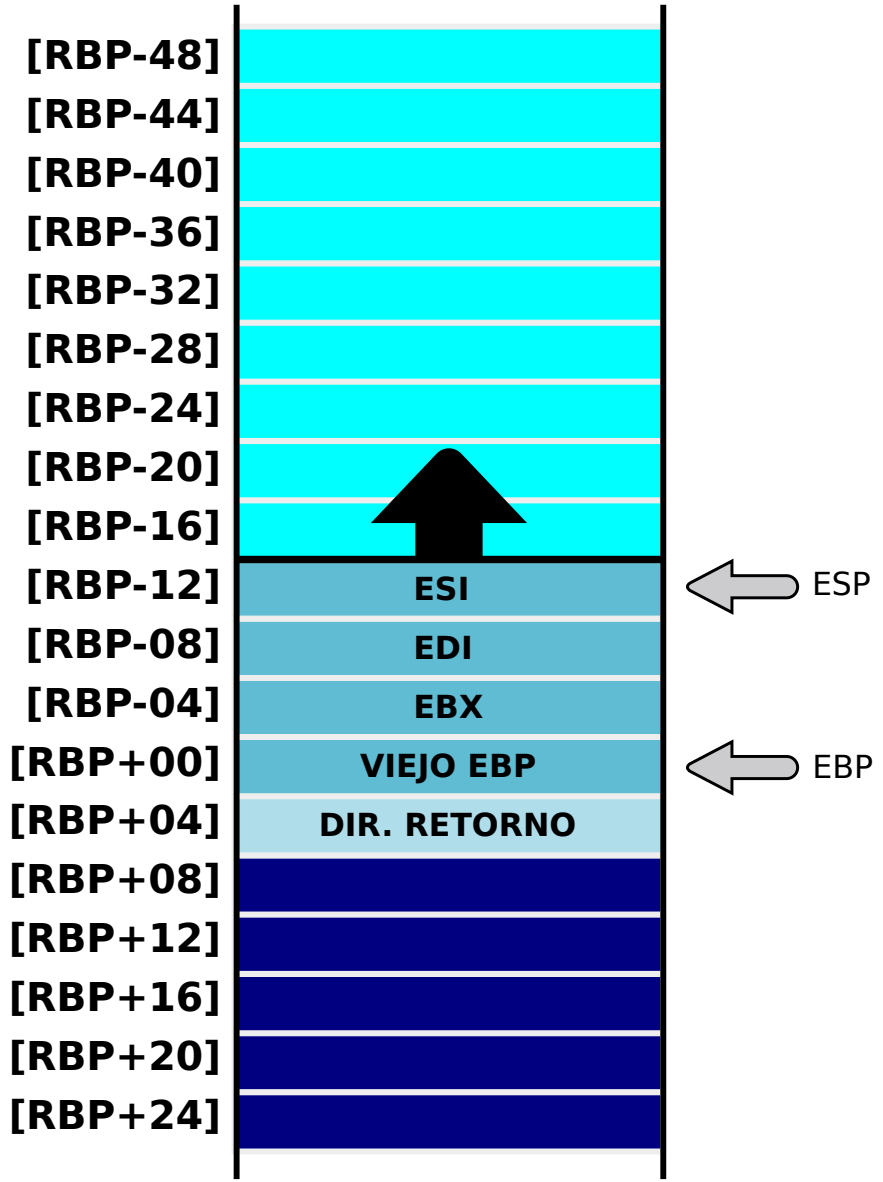
```
int fun(int a, int b){
    ...
    ...
    int res= a+b;
    ...
    return res;

}
```

Compilamos ambos programas y generamos el ejecutable de ASM:

- `nasm -f elf64 main.asm -o main.o`
- `gcc -c -m64 funcion.c -o funcion.o`
- `gcc -o ejec -m64 main.o funcion.o`

Esquema de un stack frame en 32 bits

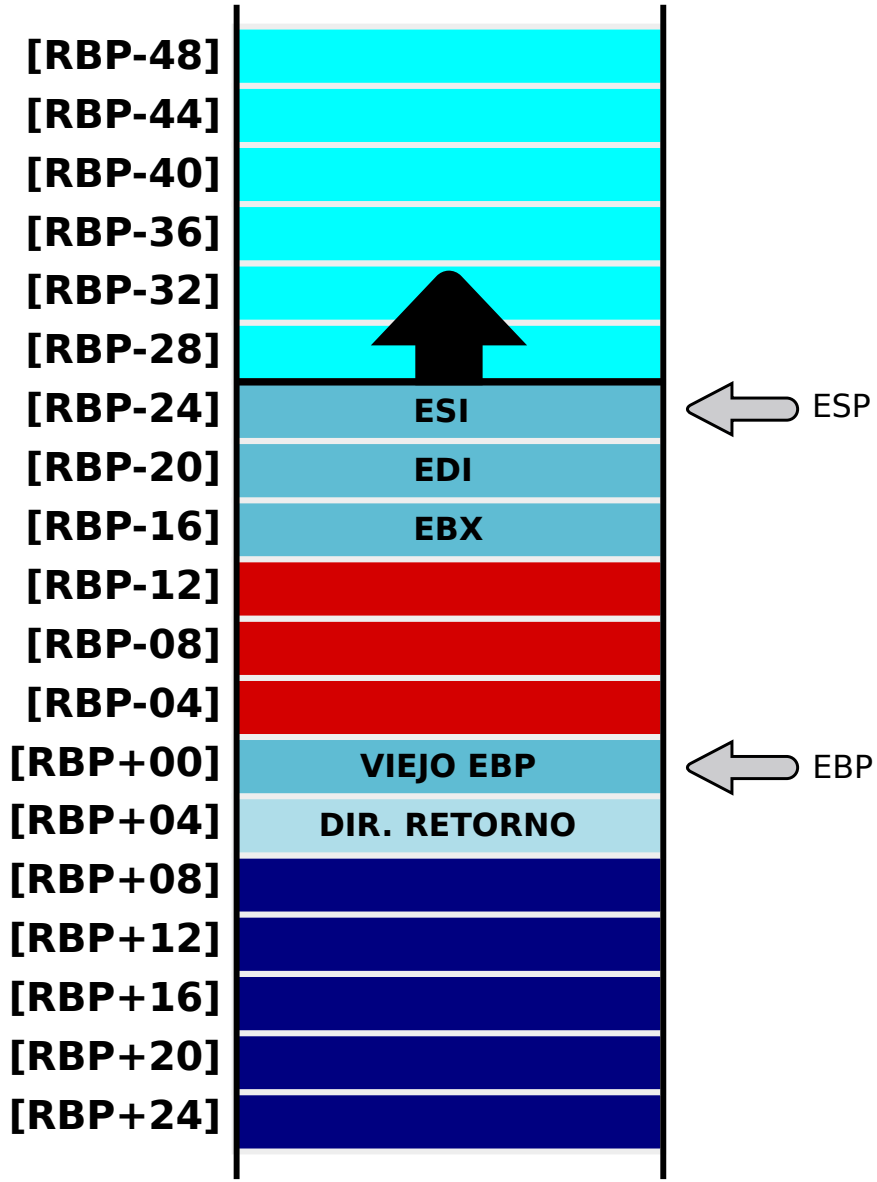
```
fun:
   push ebp
   mov ebp, esp
   sub esp, 12
   push ebx
   push edi
   push esi
   ... más código ...
   pop esi
   pop edi
   pop ebx
   add esp, 12
   pop ebp
   ret
```



Esquema de un stack frame en 32b con variables locales

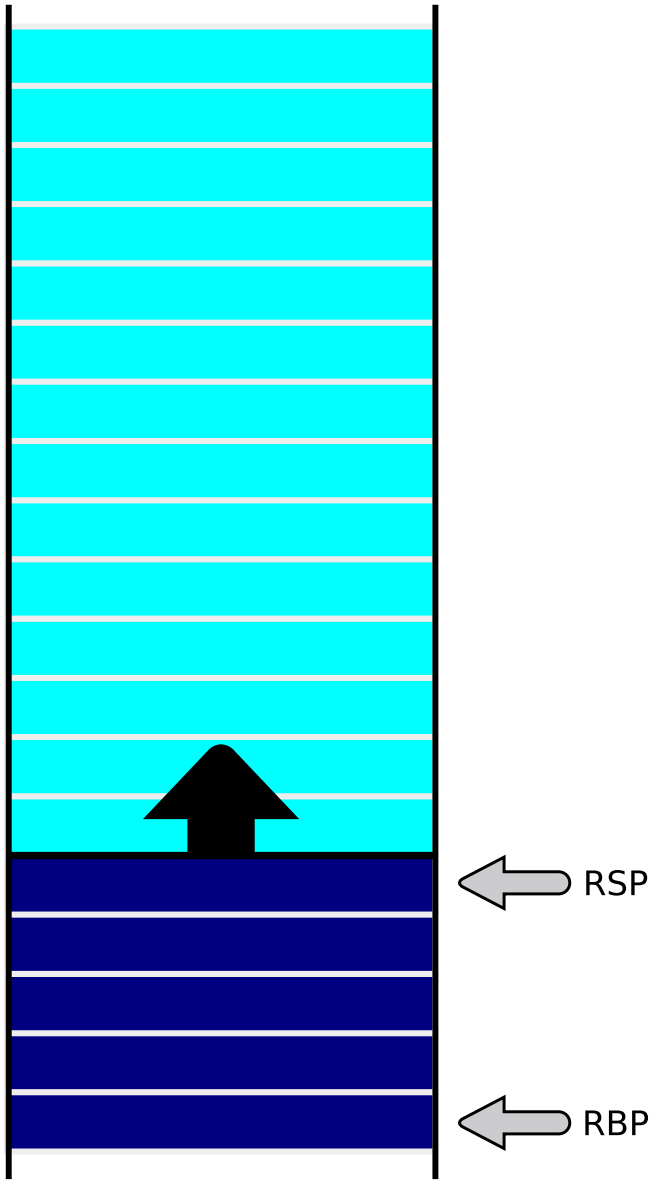
```
fun:
   push ebp
   mov ebp, esp
   sub esp, 12
   push ebx
   push edi
   push esi
   ... más código ...
   pop esi
   pop edi
   pop ebx
   add esp, 12
   pop ebp
   ret
```

Nota: El espacio a dejar depende de la alineación de la pila y las variables locales de la función



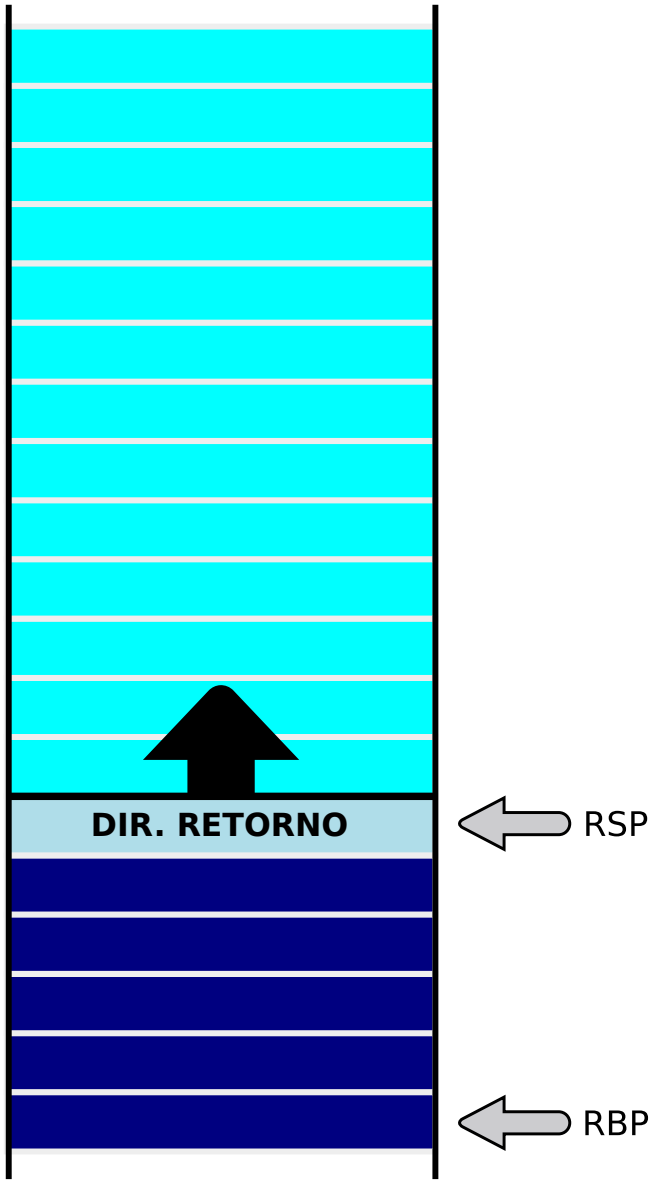
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



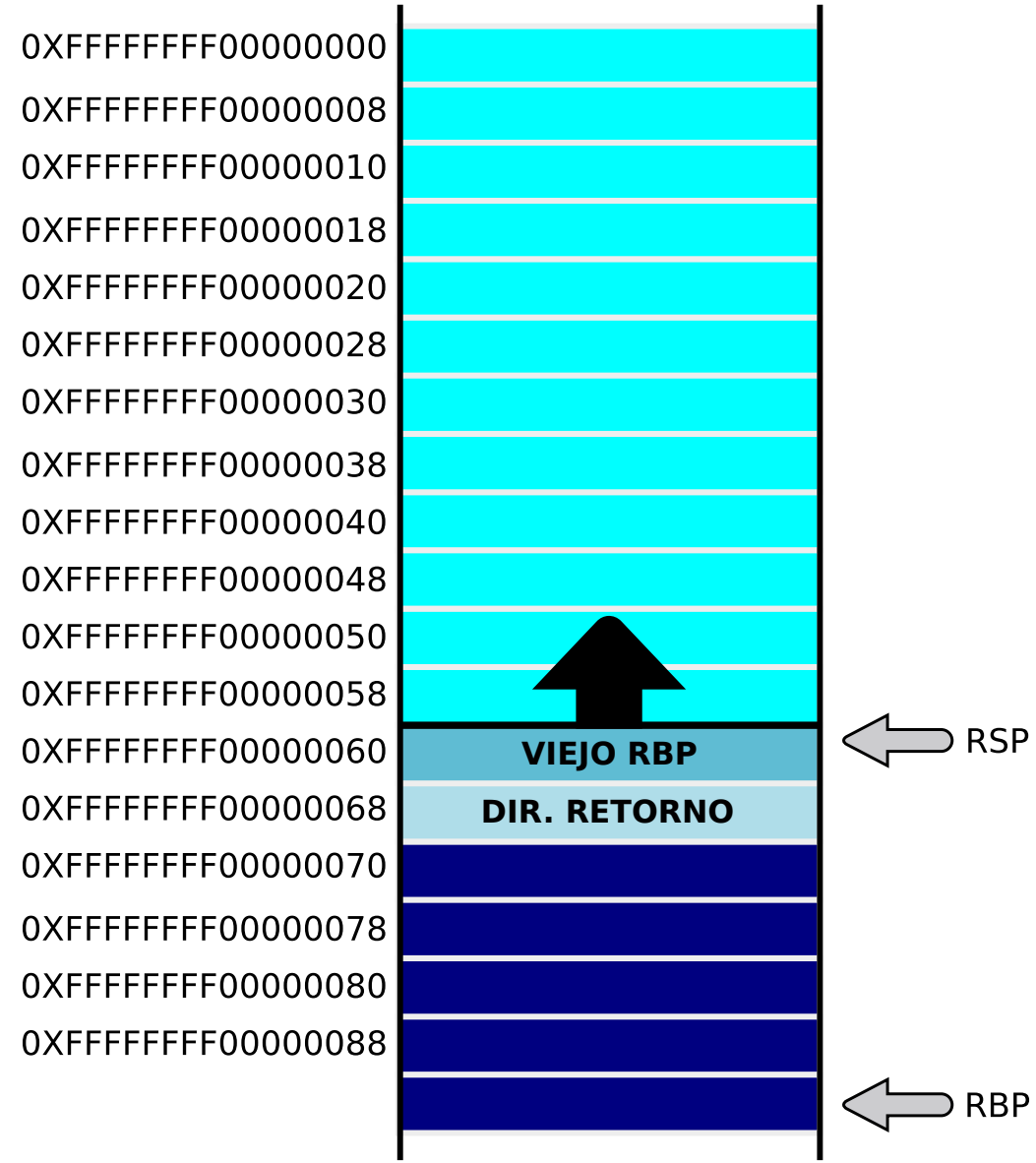
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



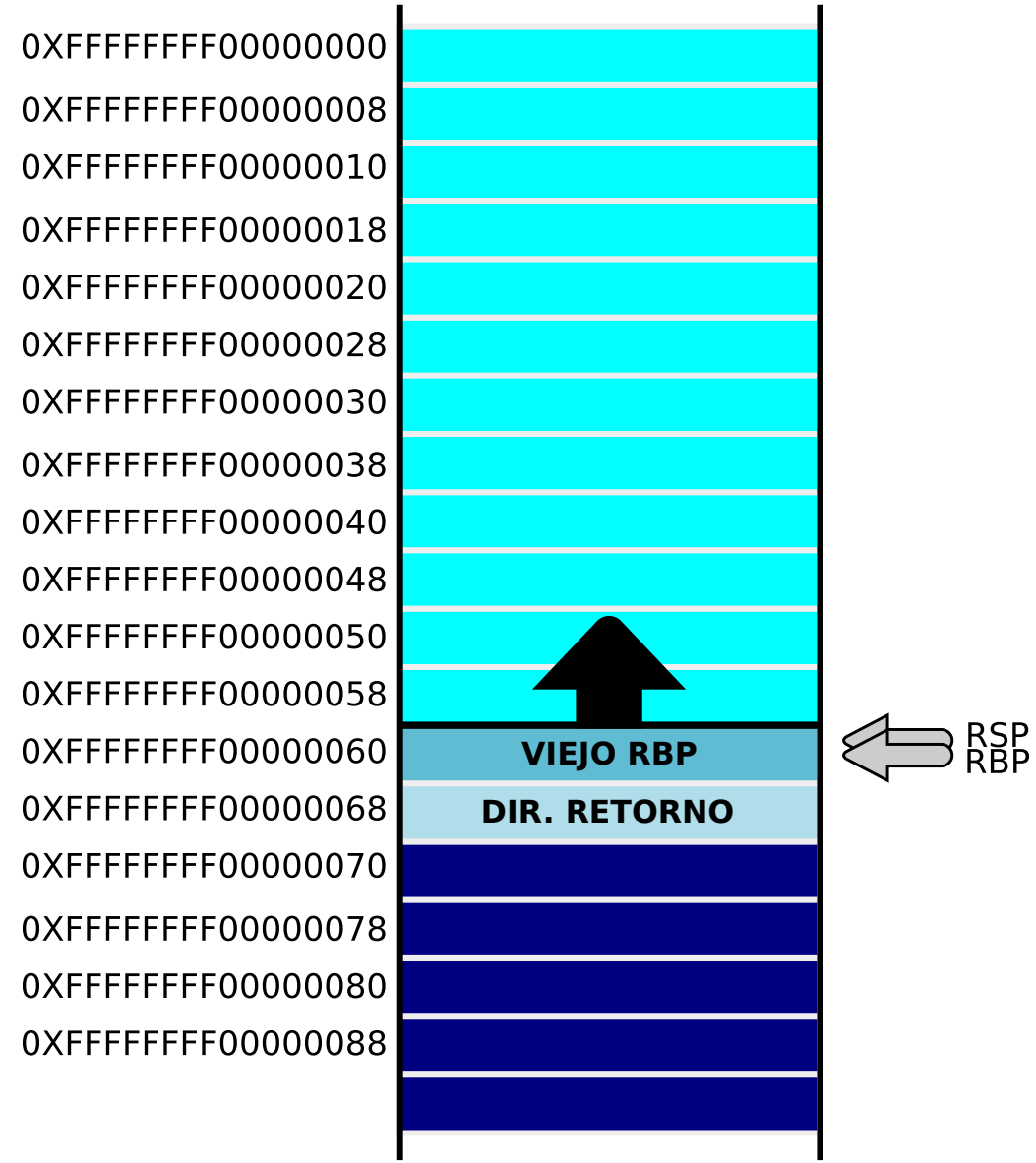
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



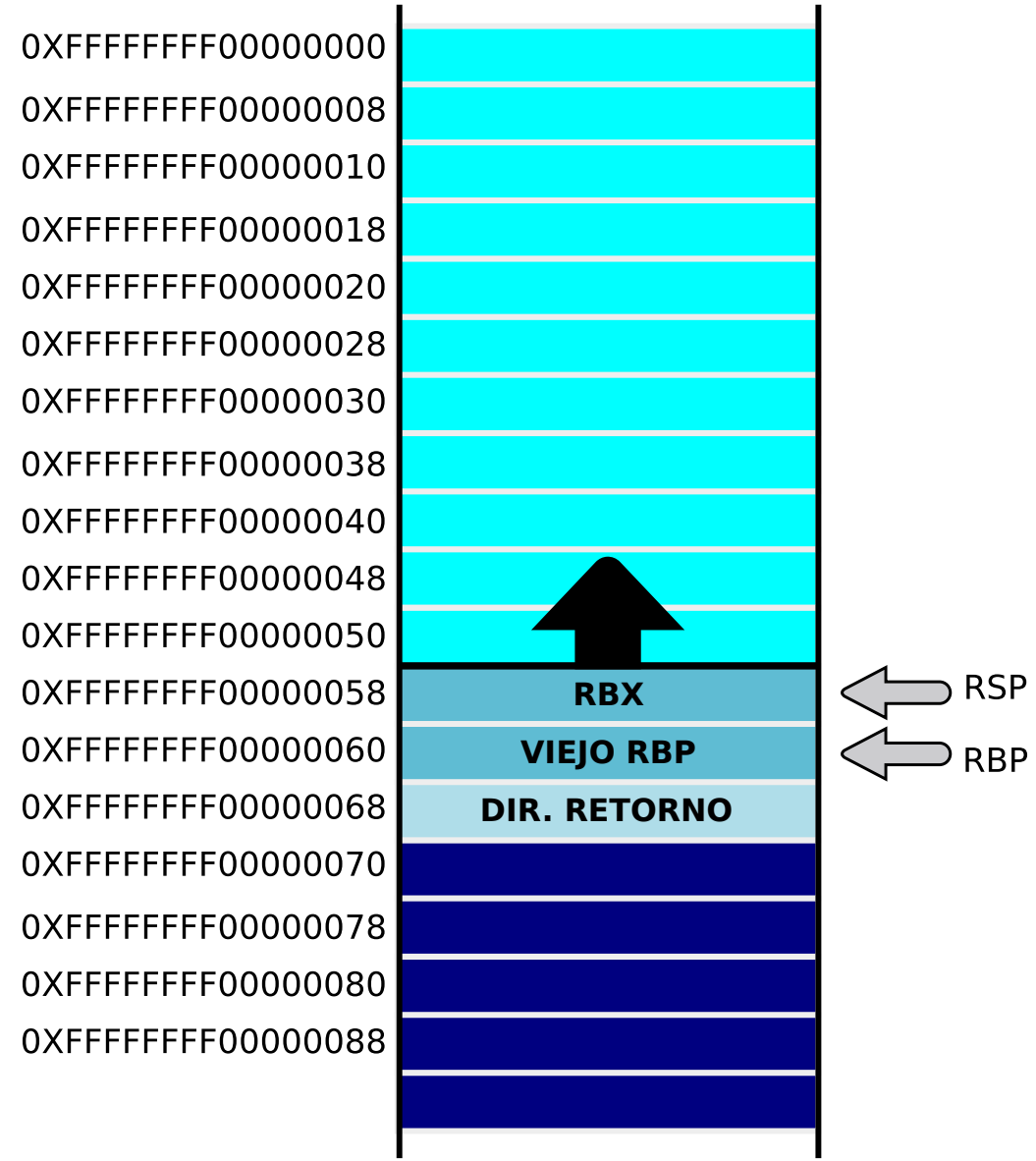
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



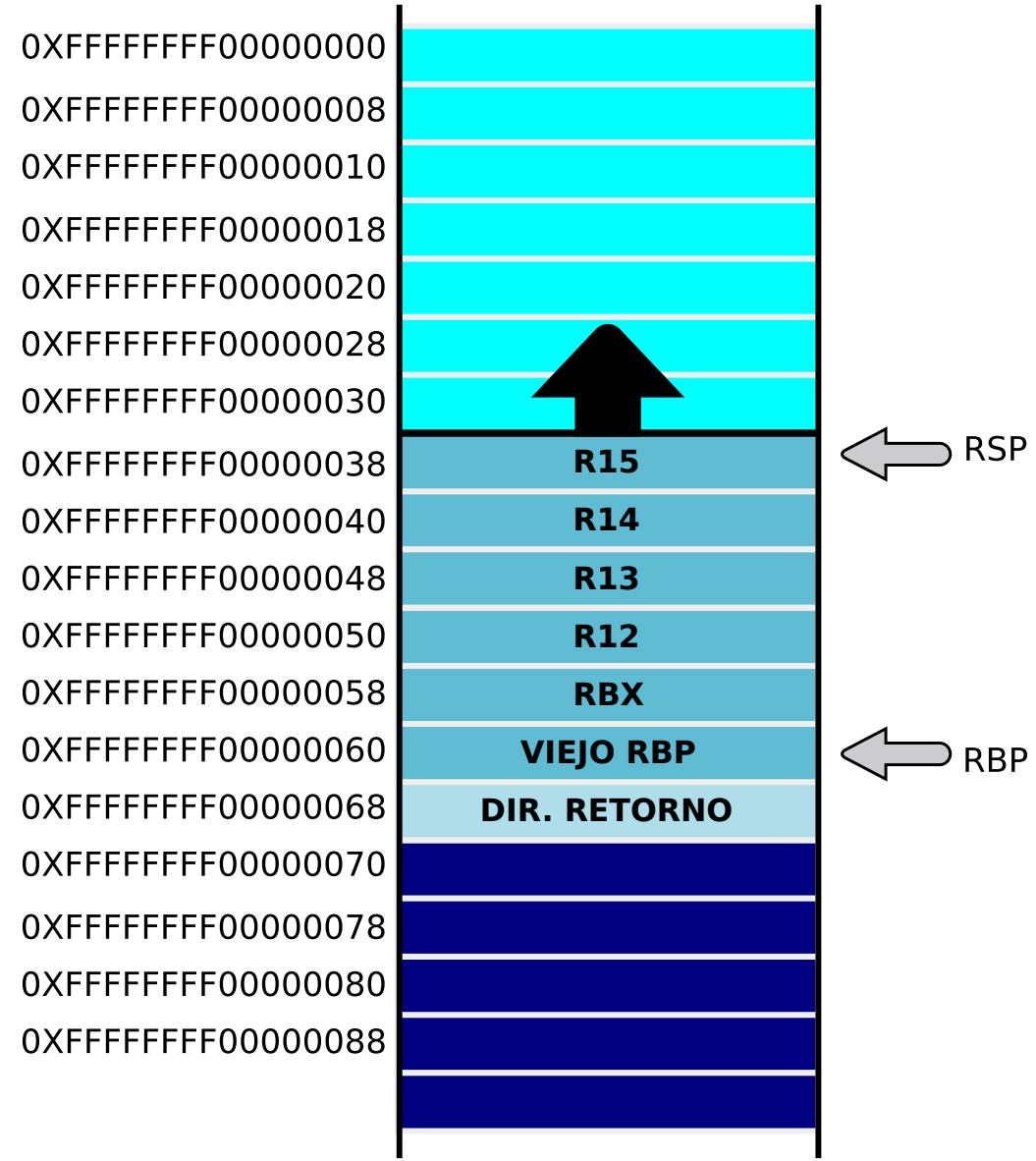
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



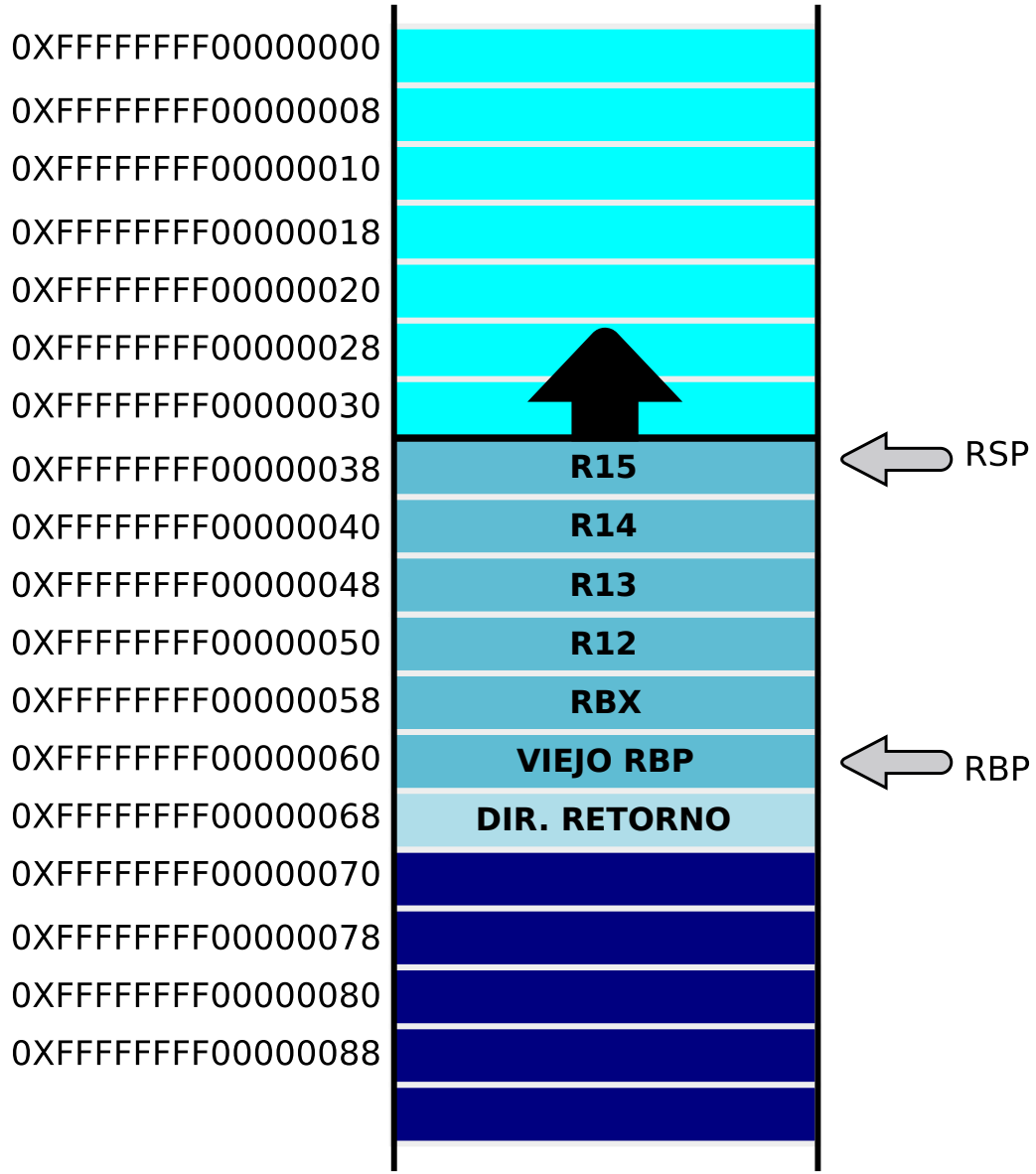
Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop r15
   pop r14
   pop r13
   pop r12
   pop rbx

   pop rbp
   ret
```



Esquema de un stack frame en 64 bits

```
fun:
   push rbp
   mov rbp, rsp

   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop  r15
   pop  r14
   pop  r13
   pop  r12
   pop  rbx

   pop  rbp
   ret
```



Variables Locales

- Una vez dentro de la función invocada en un programa C utilizamos por lo general variables locales. Solo tienen validez dentro de la función en la que se las declara.
- Una vez finalizada esta función no existen mas.
- Se crean frames en el stack para albergar dichas variables. Simplemente moviendo esp hacia el fondo del stack, es decir:

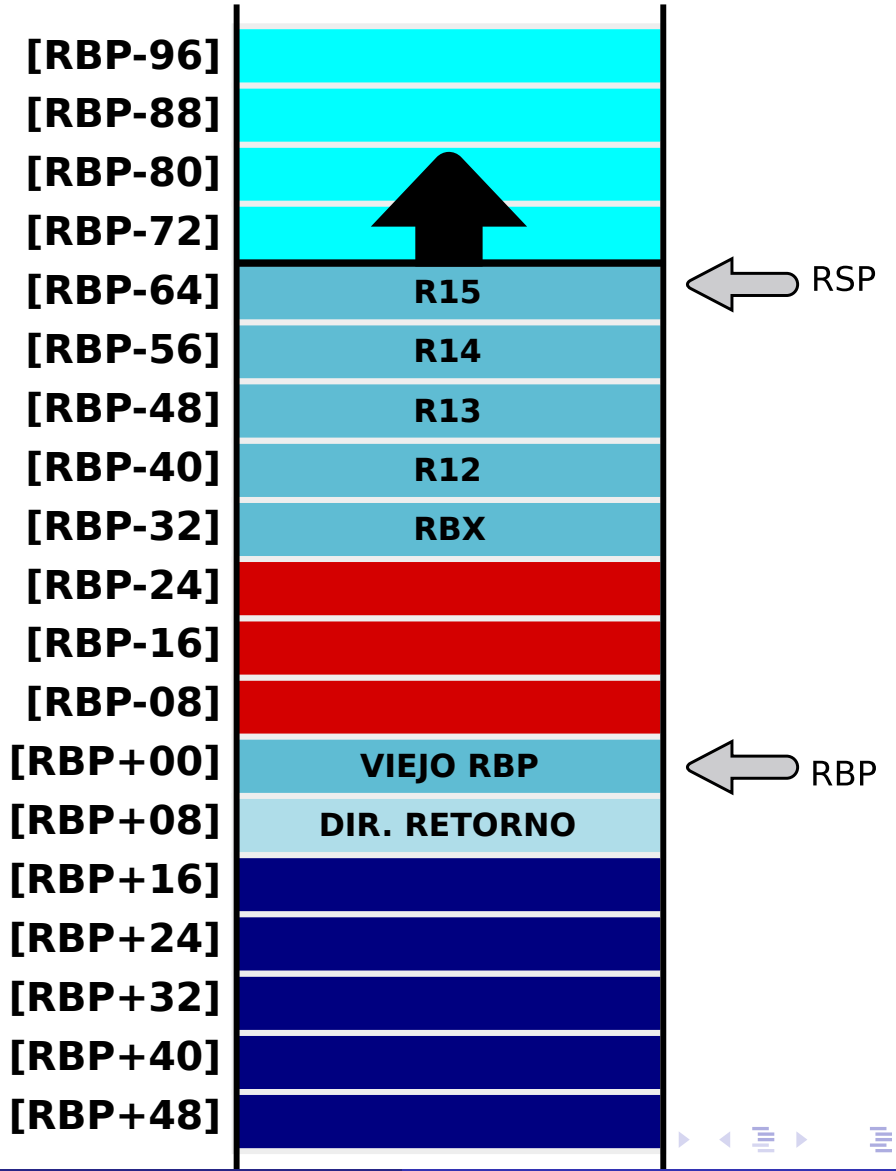
```
sub  rsp ,n
```

- Siendo n la cantidad de bytes a reservar para variables

Esquema de un stack frame en 64b con variables locales

Dejamos un espacio en la base de la pila para almacenar las variables

```
fun:
   push rbp
   mov rbp, rsp
   sub rsp, 24
   push rbx
   push r12
   push r13
   push r14
   push r15
   ... más código ...
   pop r15
   pop r14
   pop r13
   pop r12
   pop rbx
   add rsp, 24
   pop rbp
   ret
```



System V Application Binary Interface

- Conocida como ABI, establece el pasaje de argumentos desde una función llamante a una función llamada, y como se retornan los resultados.

En 64bits:

Entero o Puntero	RDI	RSI	RDX	RCX	R8	R9	[RBP+16]	[RBP+24]	[RBP+32]	[RBP+40]	[RBP+48]
Flotante	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7			

- Los registros se usan en orden dependiendo del tipo
- Los registros de enteros guardan parámetros de tipo Entero o Puntero
- Los registros XMM guardan parámetros de tipo Flotante
- Si no hay más registros disponibles se usa la PILA
- Los parámetros en la PILA deben quedar ordenados desde la dirección más baja a la más alta.

Figura: Convención C para 64 bits (©David)

Ejemplo sencillo

En 64 bits:

```
int f1( int a, float b, double c, int* d, double* e)
```

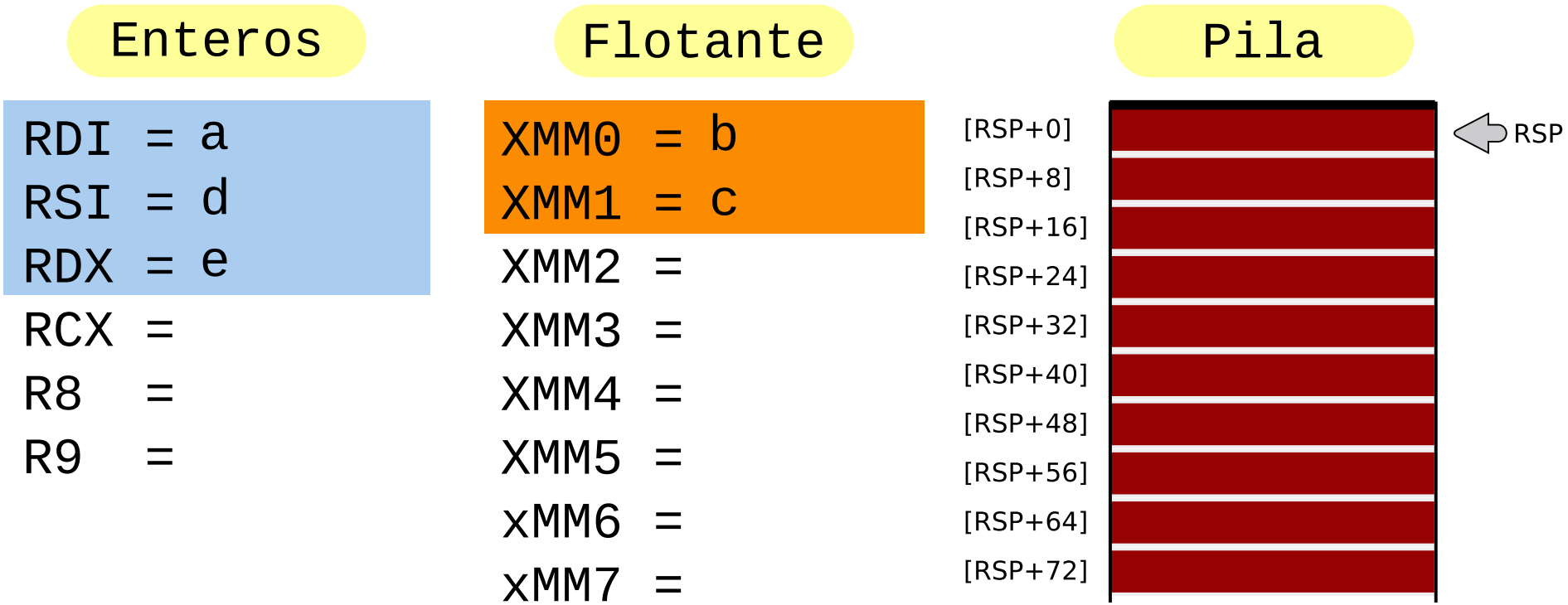


Figura: Resolución de llamada - Ej1 (©David)

Ejemplo sencillo

En 64 bits:

```
int f2( int a1, float a2, double a3, int a4, float a5,
        double a6, int* a7, double* a8, int* a9,
        double a10, int** a11, float* a12, double** a13,
        int* 14, double a15)
```

Enteros

```
RDI = a1
RSI = a4
RDX = a7
RCX = a8
R8  = a9
R9  = a11
```

Flotante

```
XMM0 = a2
XMM1 = a3
XMM2 = a5
XMM3 = a6
XMM4 = a10
XMM5 = a15
xMM6 =
xMM7 =
```

Pila

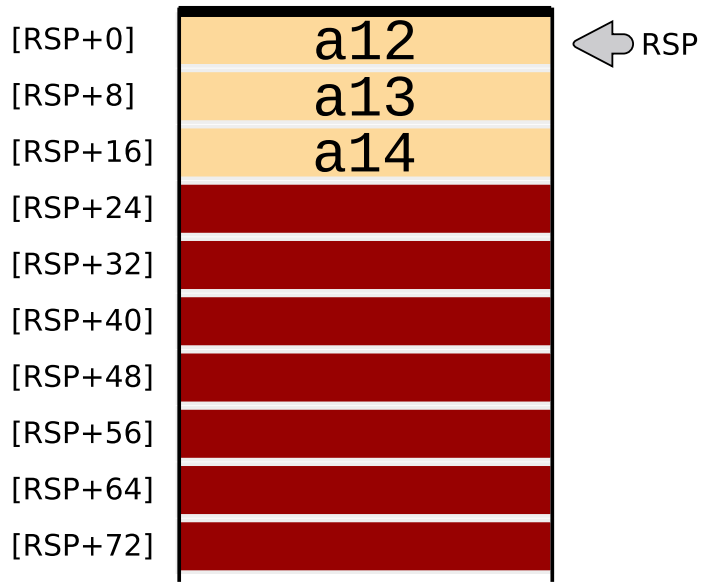


Figura: Resolución de llamada - Ej2 (©David)

Resultados

- En 32 bits los resultados enteros y punteros se devuelven por eax.
- En 64 bits los enteros y punteros se devuelven en RAX, y si son floats o doubles, en XMM0 y/o XMM1.

- Intel® 64 and IA-32 Architectures Software Developer's Manual: Vol I. Basic Architecture.

  Capítulo 6

- System V Application Binary Interface. AMD64 Architecture Processor Supplement. Draft Version 0.99.5. Edited by Michael Matz, Jan Hubička, Andreas Jaeger, Mark Mitchell. September 3, 2010.
- System V Application Binary Interface. Intel386™ Architecture Processor Supplement. Fourth Edition

Herramientas de Desarrollo

Alejandro Furfaro

15 de abril de 2020

Temario

Lenguajes

¿Que lenguaje hablan los microprocesadores?

Las CPU's definidas en los modelos originales fueron pensadas para tratar con valores que pueden tomar dos estados: Verdadero-Falso, 1 - 0, Tensión V - Tensión 0.
Por este motivo desde el inicio, cualquier Microprocesador solo "habla" en binario.
El problema es que a los seres humanos no nos resulta "natural" hablar ese lenguaje. Si bien podemos hacerlo, nos es engorroso, y por otra parte es muy fácil cometer un error. Basta con permutar un `1` con un `0` para tener un error. Y, una vez cometido,es sumamente arduo de encontrar.

Programando en el lenguaje del Microprocesador

El listado de la izquierda es el original. El de la derecha es una copia y tiene un error ¿donde está?

01101011	11011111	01101100	01101011	11011111	01101100
01000110	01110111	10001010	01000110	01110111	10001010
11101010	10010011	01101011	11101010	10010011	01101011
10100100	11010101	00110100	10100100	11010101	00110100
01100001	00010000	01101010	01100001	00010000	01101010
00011110	10001010	01011010	00011110	10001010	01011010
11010111	11010011	10100101	11010111	11010011	10100101
10001001	10010111	10011000	10001001	10010111	10011000
10001101	10100101	01111001	10001101	10100101	01111001
11000010	10010110	01101011	11000110	10010110	01101011
10110011	00101001	01111111	10110011	00101001	01111111
00101001	00010100	01101101	00101001	00010100	01101101
01010110	10010100	01100101	01010110	10010100	01100101

Programando en el lenguaje del Microprocesador

Y ? . . . ¿lo encontraste?
mmmm..... . ¿estás seguro?

01101011	11011111	01101100	01101011	11011111	01101100
01000110	01110111	10001010	01000110	01110111	10001010
11101010	10010011	01101011	11101010	10010011	01101011
10100100	11010101	00110100	10100100	11010101	00110100
01100001	00010000	01101010	01100001	00010000	01101010
00011110	10001010	01011010	00011110	10001010	01011010
11010111	11010011	10100101	11010111	11010011	10100101
10001001	10010111	10011000	10001001	10010111	10011000
10001101	10100101	01111001	10001101	10100101	01111001
11000**0**10	10010110	01101011	11000**1**10	10010110	01101011
10110011	00101001	01111111	10110011	00101001	01111111
00101001	00010100	01101101	00101001	00010100	01101101
01010110	10010100	01100101	01010110	10010100	01100101

Necesitamos un lenguaje mas “humano”

```
GLOBAL main
EXTERN printf
; Constantes
LF      equ 0xA         ; 10 decimal
CR      equ 0xD         ; 13 decimal
NULL    equ 0           ; NULL
; Datos de lectura escritura
SECTION .data
zHola   db 'Hola Mundo', LF, CR, NULL
; Codigo
SECTION .text
main:
        push dword zHola; pusheamos direccion de zHola
        call printf     ; llamamos a printf
        add esp, 4      ; ajustamos la pila
        mov  eax, 1     ; Nos preparamos....
        int  0x80       ; y nos vamos. Good bye
```

1º paso: Una sentencia = una instrucción

- Este es el lenguaje llamado Ensamblador, también conocido como “lenguaje de máquina”.
- Cada instrucción tiene un nombre alusivo a la operación que realiza (en inglés), y se lo representa por su abreviatura. Ej: MOV, por MOVE, ADD por ADDITION, etc.
- Cada sentencia en el programa corresponde a una y solo una instrucción de la CPU.
- Con ayuda de un programa llamado Ensamblador (o Assembler, igual que el lenguaje), se convierte ese texto, apto para su entendimiento por parte de los seres humanos, a números binarios, único lenguaje que habla el Microprocesador.
- Al texto original del programa escrito en lenguaje “humano” se lo conoce como ***código fuente***.

2º paso: Una sentencia = varias instrucciones

- A diferencia del Assembler, cada sentencia del programa se compone de varias instrucciones del procesador.
- La ventaja es que permite escribir aplicaciones de mayor complejidad son menos texto.
- El programa se escribe en un archivo de texto plano, igual que un programa en Assembler.
- Con ayuda de un programa llamado Compilador se convierte ese texto a números binarios, explotando cada sentencia en una o mas instrucciones del microprocesador.
- Al igual que el caso del programa escrito en Assembler, el texto escrito en C se denomina programa fuente. Obviamente esta denominación aplica al texto de cualquier lenguaje de programación.

El mismo programa anterior escrito en lenguaje C

```c
/* Esta secuencia es para iniciar un comentario.
El comentario puede ocupar cuantas lineas quieras
Y al final .....
Esta secuencia es para cerrar un comentario */

#include <stdio.h>

int     main ()
{
        printf("Hola Mundo!!\n");
        return 0;
}
```

¿Que contiene este simple programa?

1. En primer lugar lo mas fácil. Todo texto encerrado entre `/*` y `*/`, es tratado como un comentario. Significa que el compilador no va a generar código alguno con este texto.
2. Parece poco importante ya que no genera lógica ni agrega inteligencia al programa. Sin embargo los comentarios ayudan a explicar lo que estamos intentando hacer con nuestro algoritmo. Esto contribuye a la claridad de nuestro código, lo cual permite a otras personas o a nosotros mismos, modificar, corregir un defecto, o mejorar el programa con mayor facilidad. Incluir comentarios acertados y que agreguen claridad al código se considera una Buena Práctica de Programación.

¿Que contiene este simple programa?

3. Antes de continuar, aclaremos: Un programa C, se compone de dos elementos lógicos básicos: ***funciones*** y ***variables***. Las ***funciones*** contienen sentencias que definen las diferentes operaciones que se ejecutan una a una, y las ***variables*** contienen los datos que el programa mantiene almacenados, y modificará eventualemente como consecuencia de su operación.
4. Las funciones pueden llevar el nombre que mejor nos parezca, pero hay una función "obligatoria": `main`. Un programa comienza su ejecución en el inicio de la función `main`.

¿Que contiene este simple programa?

5. `main` para organizar el trabajo llama a otras funciones que como veremos van componiendo las partes que solucionan el problema completo (esto es programación modular).
6. Las funciones invocadas por `main` pueden estar escritas en el mismo archivo del programa, en otro archivo que junto con el nuestro componen el proyecto de software, o pueden ser funciones externas a nuestro programa que están guardadas en archivos que llamaremos bibliotecas de código, ya traducidas a números binarios, es decir en el lenguaje que entiende el microprocesador.

¿Que contiene este simple programa?

7. A continuación vemos la directiva

```
#include <stdio.h>
```

que le indica al compilador que debe incluir el archivo cabecera con las definiciones de las funciones de standard input output almacenadas en la biblioteca libc.

Concepto Importante

¡*stdio.h* no contiene el código de la biblioteca!. Es un archivo **de texto** en el que solamente se declaran las funciones que componen la biblioteca para que el compilador pueda conocer la sintaxis correcta para su invocación desde los programas. La biblioteca de código está en otro archivo (binario). El código fuente de las funciones que componen esta biblioteca, tampoco está en ***stdio.h***. **No olvidar este concepto**.

¿Que contiene este simple programa?

8. Toda función puede recibir una lista de valores que se denominan ***argumentos***.
9. En el caso de `main` , en esta aplicación simple no recibe argumentos. Mas adelante en el curso veremos que puede recibirlos y como tratarlos en tal caso.
10. Luego entre los caracteres { y } se encierran las sentencias que componen el cuerpo de la función.
11. En el caso de este sencillo ejemplo el cuerpo de `main` solo contiene las sentencias:

```
printf("Hola Mundo!!\n");
return 0;
```

¿Que es printf?

12. No es otra cosa que una función.
13. Tal como explicamos recibe un argumento, en este caso el texto Hola Mundo!!\n
14. Lo que hace **`printf`** es imprimir en pantalla el texto que le pasamos como argumento.
15. \n es una secuencia de escape que utiliza el lenguaje C para representar el caracter Nueva Línea.
16. De este modo el comportamiento esperado de nuestro programa será imprimir en pantalla en el renglón siguiente al comando que lo ejecute, el mensaje Hola Mundo!!, y luego saltar a la línea siguiente como si se pulsase la tecla <Enter>
17. El tipo de argumento es una cadena de caracteres en forma de constante, por eso va encerrada entre comillas dobles.
18. A lo largo del curso vamos a utilizar mucho las cadenas de caracteres, de modo que es bueno empezar a familiarizarnos desde el principio.

¿Donde está printf?

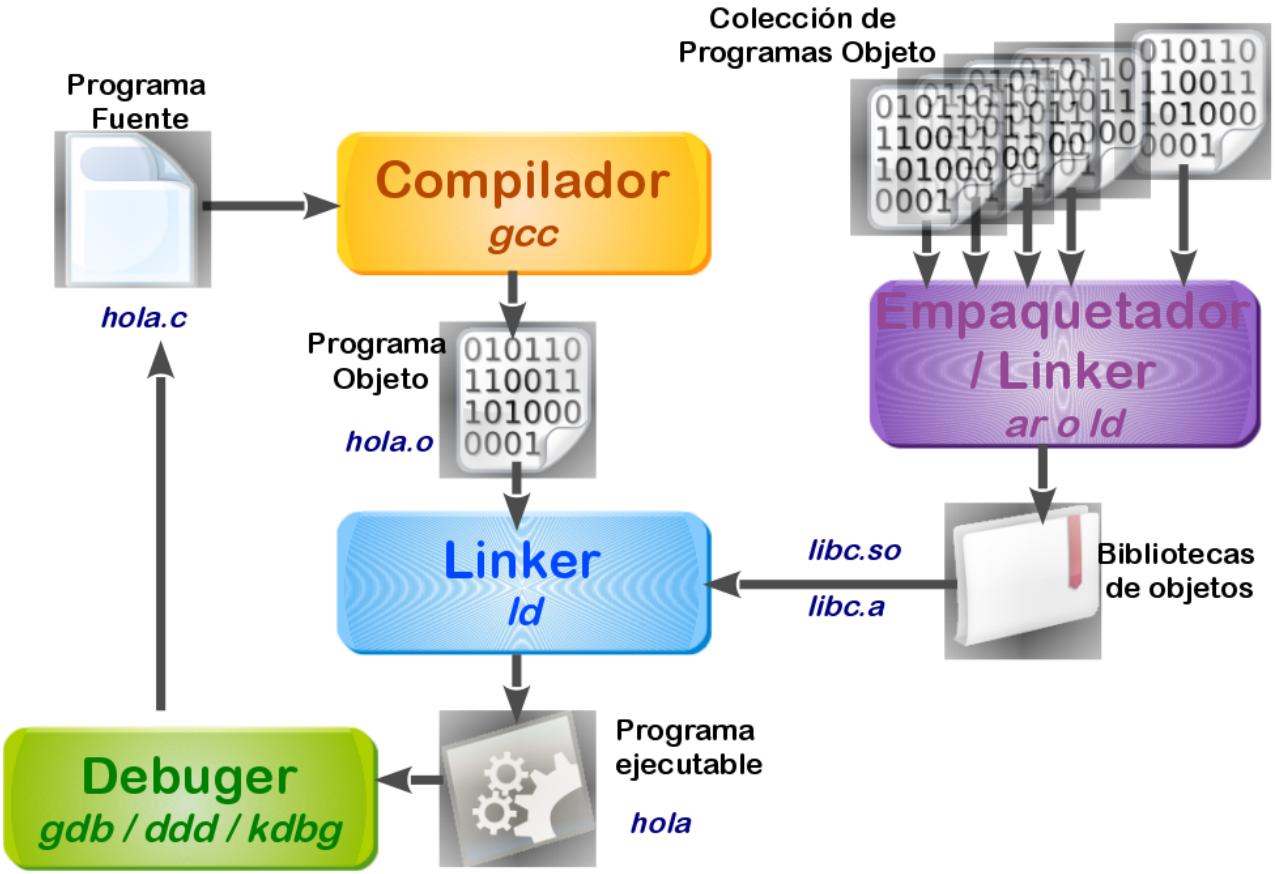
19. En nuestro archivo fuente, evidentemente no está.
20. De modo que solo cabe una posibilidad: La función es externa.
21. `printf` está contenida en una de las bibliotecas mas utilizadas en C: La de entrada salida estándar, cuyas definiciones estan en el archivo header stdio.h, ya explicado.
22. Comprobémoslo:

Tipear en la consola

```
locate stdio.h
grep ' printf ' /usr/include/stdio.h
```

23. Alguno de uds. estará preguntándose como se logra que el programa acceda al código de `printf` si ésta no es parte de programa sino que está afuera de él ¿verdad?
24. Quienes aun no se lo preguntaron... deberían hacerlo ;)

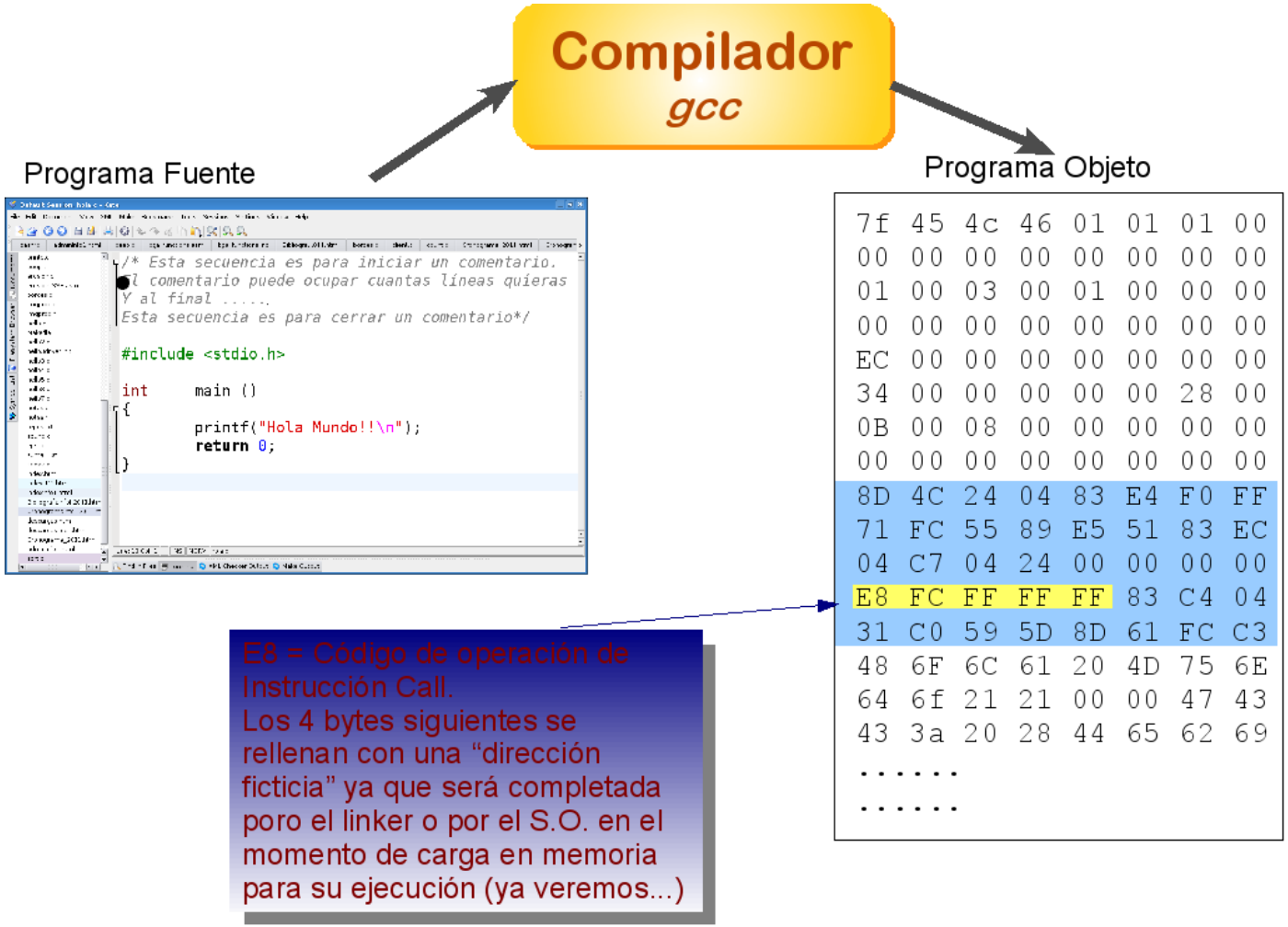
Proceso de desarrollo



El compilador

- Es un programa capaz de analizar sintácticamente un archivo de texto que contiene un programa fuente.
- Si éste está escrito de manera correcta, respetando la semántica del lenguaje para el cual compila, genera un código binario adecuado para ser ejecutado por el Microprocesador que obra como CPU en el sistema.
- Además de analizar las operaciones reemplaza los nombres lógicos que adoptemos en nuestro programa para variables o funciones por las direcciones de memoria en donde se ubican las mismas.
- No puede resolver referencias a funciones exteriores al archivo fuente que analiza. Por ejemplo, no puede resolver por que valor numérico reemplazar a la etiqueta `printf` , ya que no tiene visibilidad de la misma. Habrá que esperar a la siguiente fase para resolver este tema.

Cuando se dejan referencias por resolver



El compilador

- Antes de hacer su trabajo, invoca a un programa denominado preprocesador, que se encarga de eliminar los comentarios, incluir otros archivos (la línea #include <stdio.h>, es reemplazada por contenido del archivo stdio.h), y reemplaza las macros (la sentencia para el preprocesador en este caso es #define).
- Si genera errores el programa está mal escrito y debe ser revisado.
- Si no genera errores solo significa que el programa está correctamente escrito. De allí a que funcione correctamente es otra cuestión. . .
- Una vez que compiló, su producto es un programa objeto. Este es un binario pero que aún no está listo para poderse ejecutar.

Para generar el programa objeto, tipear en la consola

```
gcc -c hola.c -ohola.o -Wall
```

El Linker

- Es un programa capaz de tomar el programa objeto generado recién por el compilador, enlazarlo (“linkearlo”) con otros programas objeto y con otras biblioteca de código y generar un programa ejecutable por el Sistema Operativo sobre el cual estamos desarrollando nuestro programa.
- Muchas cosas juntas ¿verdad?
- Enlazar significa:
  - Poner todos los bloques de código juntos y ordenar código y datos en secciones comunes para luego guardar ese conjunto en un único archivo ejecutable.
  - Una vez ordenado, resolver cada referencia a una variable o función que en la fase de compilación eran externas. En nuestro caso el linker resolverá la referencia a `printf` .
  - Identificar y marcar el punto de entrada del programa (la dirección que se le asignará a `main`).

El linker

- Parece poco relevante. Sin embargo es crucial esta fase de la generación de nuestro programa

Para generar el programa ejecutable podríamos, tipear en la consola

```
ld --eh-frame-hdr -m elf_i386 --hash-style=both
-dynamic-linker /lib/ld-linux.so.2 -o hola /usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/gcc/i486-linux-gnu/4.3.2/crtbegin.o
-L/usr/lib/gcc/i486-linux-gnu/4.3.2 -L/usr/lib hola.o -lgcc
--as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed
-lgcc_s --no-as-needed /usr/lib/gcc/i486-linux-gnu/4.3.2/crtend.o
/usr/lib/crtn.o
```

- Hay involucrados unos cuantos objetos como vemos que son relevantes: crt1.o, crti.o, crtbegin.o, crtend.o.
- Y algún que otro componente adicional.
- Engorroso, imposible de memorizar, y sobre todo, sujeto a cuestiones internas del sistema.

El linker

- Por eso, gcc sabe llamar al linker y nos evita este engorroso trámite a nosotros

Para generar el programa ejecutable tipeamos en la consola

```
gcc -ohola hola.o -Wall
```

- Para saber como el gcc arma el llamado usamos la opción -v (verbose)

Tipear en la consola

```
gcc -ohola hola.o -v
```

Warning: Prestale atención a los warnings

- Los Warnings que arrojan tanto el Compilador como el linker, son commo su nombre lo indica Ardvertencias.
- No impiden que el compilador genere el archivo con el objeto reubicable ni que el linker genere el archivo ejecutable.
- No por eso debemos ignorarlos.
- Por el contrario debe prestarse especial atención a los Warnings
- La experiencia indica que terminan transformándose en errores de lógica.
- La opción `-Wall` en ambas líneas de compilación y linkeo, indica a ambas herramientas que presenten Todos los Warnings (`W`arning `all`). Aun los mas insignificantes

Buenas Costumbres

Buena Práctica de Desarrollo

Siempre incluir la opción `-Wall` en las líneas de compilación y linkeo, trabajando sobre el código para eliminar **TODOS** los warnings.
Algo es 100 % seguro: ***El Warning se transforma en un bug mas tarde o mas temprano***.
Por lo tanto se deberá incluir en cada proyecto `-Wall` en las líneas de compilación y linkeo **SIEMPRE**.

Agreguemos alguna función de cálculo

```
/* Programa sqrt.c:
*  Su funci n es calcular la ra z cuadrada de un n mero
*  predefinido en su c digo y mostrar su resultado en
*  la pantalla del computador.
*  Para compilarlo: gcc -c sqrt.c -o sqrt.o
*  Para linkearlo: gcc sqrt.o -o sqrt -lm
**/

#include <stdio.h>
#include <math.h>

#define N 1234567890

int main ()
{
        double result;
        result = sqrt(N);
        printf ("La raiz cuadrada de %d es:%10.7f\n",N,result);
        return 0;
}
```

Linkeando con una Biblioteca

- Si observamos el comentario que encabeza el listado del programa del slide anterior, vemos que al linker se le provee una opción adicional: `-lm`
- `-l` sirve para especificar el nombre de una Biblioteca (l por library)
- m es el nombre de la biblioteca: `m` es math, cuyos prototipos, macros y constantes están definidos en math.h (entre ellos la función `sqrt`)
- Pregunta: ¿Porque no hubo que especificar la librería que contiene `printf` ?
- El compilador “conoce” la ubicación de las bibliotecas mas comunes para evitar que debamos especificar permanentemente librerías de uso casi tan común como la propia función `main`

Que Aprendimos?

- Que son y que relación tienen los diferentes lenguajes, binario, assembler, C.
- Las herramientas de desarrollo que utilizamos para construir programas, su uso y conceptos.
- Hicimos algunos ejemplos para empezar a caminar.
- Ahora vamos a mejorarlos y aumentar sus posibilidades